(2)

Productivity Engineering in the UNIX[†] Environment

SimPL: A Simulator for Parallel LISP

Technical Report

S. L. Graham
Principal Investigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

~~87 9 4 033~~

SimPL: A Simulator for Parallel LISP

Lisa Penninger

Computer Science Division
Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

June 2, 1987

Abstract

SimPL is a simulator for a multiprocessing LISP that incorporates both delayed and eager-beaver style evaluation; it runs on a single processor. SimPL can be used to run a variety of programs with different amounts of parallelism. The results of these simulations can be used to make generalizations about the benefits gained from use of parallelism in certain symbolic computations, and, given other data about process-starting time, the optimal number of processors for this type of computation. Some examples from symbolic mathematical computation are used to illustrate the program.[1]

[1]Sponsored in part by the Army Research Office (grant DAAG29-85-K0070) through the Center for Pure and Applied Math and DARPA/XCS (grant 25857).

Table of Contents

Table of Figures

SimPL: A Simulator for Parallel LISP

Lisa Penninger

Computer Science Division
Electrical Engineering and Computer Science
University of California
Berkeley, CA 94720

1. Introduction

In spite of the great interest in speeding up programs by using parallelism, few convenient parallel machines are available, and those may not have the types of parallelism that might be of most benefit. This paper describes a simulator, written in COMMON LISP, that provides a method for experimenting with varying amounts of explicit (programmer-specified) parallelism, and gathering information about the efficiency of such programs if they were to be run on a parallel processor.

The system chosen for simulation is Multilisp [7,8]. Its parallel constructs are quite simple, and rely heavily on the programmer to decide on their efficient and correct use. Multilisp is explained in a later section of this paper; however, a working knowledge of some dialect of LISP is assumed.

1.1. Related Work

There is a history of partially implemented or one-of-a-kind machines and systems to run parallel LISP. See, for example, references on Concurrent LISP [14], Butterfly LISP [13], SPUR Lisp [16], Concurrent Common LISP [2] and the Bath Concurrent Lisp Machine [11]. Two models that seem to have emerged, with variants, are Q-LISP [5] and Multilisp [8].

Queue-based multiprocessing LISP, or Q-LISP [5], allows run-time specification of the type and amount of multiprocessing done. Its parallel constructs, `qlet` and `qlambda`, have similar syntax to `let` and `lambda`, but take an additional argument, which specifies whether they are to execute sequentially, in parallel but waiting for values from the other arguments, or in parallel without waiting. An interesting feature not present in other languages is the ability to prematurely terminate a process chain and return a value. This is done through modified versions of `catch` and `throw`, thus making this LISP useful for OR parallelism, which is used in AI applications such as search trees. A simulator has been written, and the language is currently being implemented on an Alliant multiprocessor.

The Multilisp system is discussed in more detail in a later section of this paper. It is implemented on the 32-processor Concert system.

In spite of these systems, a simulation provides the flexibility of running programs on un-realized architectures, although at slow speed. It is easier to change software than hardware.

Another COMMON LISP simulation package has been implemented by McGehearty and Krall [12]. This package differs from SimPL in that it uses a preprocessor to insert function calls to handle synchronization of the simulated processes, rather than a post-processor. The synchronization handlers output timing data when the program is run. Then a post-processor analyzes the data in the same manner as SimPL.

1.2. Outline

Section 2 discusses the categorization of various types of parallel systems. Section 3 describes Multilisp. Section 4 discusses the proper use of high-level explicit parallelism in programs. Section 5 describes SimPL. Section 6 gives a detailed description of the implementation of SimPL. Section 7 describes how to use SimPL, the programs that were run, and the results. Section 8 summarizes the work, and explores directions for further research.

2. Description of Multilisp

Multilisp [7,8] is a LISP dialect that is based on Scheme [1] and has been extended by the addition of the following four parallel constructs. These constructs, which have been adapted for use in this simulator, are the only method of introducing parallelism into Multilisp programs. Unless the programmer explicitly specifies them, the program will execute sequentially.

2.1. Delay

Delays are an implementation of the concept of *lazy evaluation* where the value of an expression is not computed until it is required. Therefore, delays may be thought of as examples of demand-driven data flow programming. An expression which requires the value of a delay is data-driven; it executes only after its arguments are available. The construct

```
(delay A)
```

returns a structure which contains all information necessary to evaluate the expression A at a later time. Evaluation will not begin until the value of A is actually required. When this occurs, the delay is computed on the same processor on which its value was required, since the original process is blocked until it receives that value. Therefore, delays affect the order of execution of the program, but do not affect parallelism.

2.2. Future

Futures are based on the *eventual values* developed for Algol 68 on the C.mmp system [10]. The construct

```
(future A)
```

causes its parent process to be pushed onto the current processor's stack, while the current processor begins evaluation of the expression A. The parent process may be removed from the stack and executed by any free processor. It is able to continue execution because the future call will immediately return a structure which acts as a placeholder. This structure contains information about the process evaluating A, including whether its value has been computed yet, and will eventually hold A's return value. A future which has finished evaluation is said to be *determined.*

2.3. Touch

The construct

```
(touch A)
```

behaves as the identity operator, unless its argument is an undetermined future or delay, in which case its execution is suspended until the future or delay has been determined.

2.4. Pcall

In a sequential LISP system, the arguments to a function call are evaluated in turn. Multilisp allows the arguments to be evaluated in parallel, by use of the construct `pcall`. Evaluation of the form

```
(pcall F A B C ...)
```

results in the expressions F, A, B, C, ... being evaluated in parallel to produce the values f, a, b, c .... When all have been evaluated, f (presumably a function definition) is applied to a, b, c, ... to compute the value of the expression.

This can be implemented by creating futures to compute each of the values, and then *touch*ing each one to force its evaluation before the function is called.

2.5. Implementation

The structure that Multilisp uses to represent a future is considered a separate type. It contains a LISP value, a task queue, a "determined" flag, and a lock. The LISP value will be used to contain the final value of the future. The task queue is a list of all processes which are blocked because they require the future's value. The determined flag is true if the computation of the future's value is finished. Initially the flag is false and the queue is empty.

Multilisp evaluates function calls in the same way that a sequential LISP system does, that is, left-to-right. It checks the type of each object to see if it is a future structure. If not, it simply returns the LISP value of the object. Otherwise, if it is a future structure, and the future is determined, then the LISP value is returned. This value must be recursively checked to see if it is also a future. The only case in which a process blocks waiting for the value is if the future is undetermined and its value is required immediately. The process will then be suspended and added to the task queue of the future. When the future finally does become determined, these tasks will be activated.

Once a future is determined, the structure used to represent it no longer serves any purpose, and the Multilisp garbage collector will substitute the value for the structure, thus eliminating a level of indirection.

When Multilisp evaluates an expression whose car is future, the processor will push the original task onto its queue of waiting tasks and proceed with the evaluation of the parallel construct. The original task may be picked up by any idle processor; if there are no idle processors, then evaluation will proceed as if on a sequential system.

Similarly, when a processor evaluates

```
(pcall + A B C),
```

the originating processor evaluates the expression A and pushes the expression B onto the queue associated with the processor. B will be selected for evaluation after A is finished, or else evaluated on some other processor.

3. Categories of Parallelism

Parallel systems can be divided into classes along several lines. These distinctions are important to consider when choosing or implementing a parallel system; some classes are more appropriate for certain applications. problems.

3.1. Bounds on Resources

If an unlimited number of processors were available for solving a problem, then the system would be *unbounded.* In real life, this is not the case -- systems have a fixed number of processors; they are *bounded.* The question to consider with bounded parallelism is how to best use the processors available, while unbounded parallelism is primarily useful for theoretical computation of the lower bound on execution time of a particular problem [15].

3.2. Complexity of System

The complexity of a system defines another class boundary. The number of instruction streams (processors) and data streams involved are usually divided into classes of "single" and "multiple". These two divisions are combined to describe the system, which results in four possible arrangements: Single Instruction-stream, Single Data-stream (SISD), Multiple Instruction-stream, Multiple Data-stream (MIMD), Single Instruction-stream, Multiple Data-stream (SIMD), and Multiple Instruction-stream, Single Data-stream (MISD). A example of a

SISD system is the traditional von Neumann machine. MIMD systems are the most general type, with multiple processors operating on multiple data objects. An array processor would be an example of a SIMD machine, where a single program performs the same operations on many data objects at once. MISD systems aren't generally used, but an example is a pipelined architecture, where several processors operate in sequence on a stream of data, each taking as its input the previous processor's output.

3.3. Level of Parallelism

When the primitives of a system implement the parallelism by overlapping machine instructions presented in a sequential stream, the system is said to employ *low-level* parallelism. An example is a pipelined CPU, where certain parts of each operation are allowed to proceed in parallel. When the algorithm itself is specified in a parallel fashion, for example, as the result of combining several distinct processing streams with occasional interlocks, then *high-level* parallelism is being used. Multilisp is an example of the use of yet higher-level parallelism, although the programmer still must specify where the parallelism can occur, the precise ordering of processing and interlocks may not be specified.

3.4. Strategy for Problem Solving

In *AND* parallelism, the problem to be solved is divided into several tasks executed in parallel, and all results are required for completion. An example is computing the arguments to an addition function in parallel, since both of the values are required for the addition to finish.

In *OR* parallelism, only one of the tasks must be completed. The other processes may then be terminated. An example is searching the branches of a game tree in parallel. If one of the branches shows a win, then it is not necessary to finish searching the others. Tasks which pursue alternative strategies in this way are called *colluding* processes [9]. Unlike AND parallelism, OR parallelism is also useful on sequential multitasking systems, as the overhead of scheduling processes can be offset by the time saved if one process finishes much sooner than the others.

3.5. Flow of Control

Traditional von Neumann programs consist of a series of instructions which execute in the order specified by the programmer. However, another model exists, called *data flow* computation A data flow program consists of a group of instructions whose order of execution is determined by the interdependencies of the data they operate on. There are two divisions of data flow programs: data-driven and demand-driven. A data-driven instruction will execute when its operands are ready; a demand-driven instruction will execute when its result is required.

3.6. Method of Communication

Two methods of communication between processes are the use of *shared memory* with *monitors* and *message-passing*. A monitor is a data structure or a collection of structures together with routines to modify it, e. g. a queue with pop and push routines. Message-passing involves sending copies of data objects to other processors in order to communicate. This may be done with blocking, i. e. when a process sends a message it blocks until the recipient is ready to receive it, or with buffers or queues, so that a process may send its message and go on with other computation without waiting. Several issues must be considered when designing this type of system:

(1) Do processes automatically block when sending or receiving messages, or is there some other method of synchronization?

(2) How do processes know how to contact others they wish to communicate with?

(3) What is the format of a message?

(4) How are communications failures handled?

3.7. Type of Symmetry

Typical programs use subroutines, in which the caller must know the name of the callee, but the callee does not know which routine has called it. This is a hierarchical, or *asymmetric* transfer of control. An example of *symmetric* transfer would be a program which used *coroutines* [3]. Coroutines are like subroutines, except that they do not simply return to the next instruction in the calling procedure. They pass control to each other always by name, and may have many entry and exit points.

In a parallel environment, the caller of a subroutine may possibly go on executing without waiting for the return value. A synchronization point occurs when it reaches a point where it cannot continue without that value. Coroutines can also be extended to work in the parallel world; a coroutine may execute until it requires some kind of communication with some other routine. At that point, either the other routine is ready and waiting for communication, or the first routine may have to wait until the other routine also reaches the point where it is ready to communicate. After that point, they may both continue.

4. Placement of Parallelism

In some parallel systems, the parallelism is implemented transparently by the hardware and/or software (implicit parallelism). In others, the programmer must specify in the program itself which operations may be done in parallel (explicit parallelism). Still others use a combination of the two.

Explicit parallelism can cause programmer headaches in several ways. In a shared memory system, one must consider whether parts of the program will interfere with each other if processed in parallel. The overhead involved in creating a new process is important; if it exceeds the time saved by the offloading, it is better to execute the computation in-line on the same processor. Another consideration is how soon the result will be needed. If it is required by the originating process immediately, then the originating process will be blocked until the result is received. The task could have been computed on the original processor with two benefits: saving the overhead of sending the task to some other processor, and freeing the second processor for other work.

4.1. Definition of Concepts

Three properties are useful for determining where to use parallelism for maximum speedup. A function is *strict* if the values of its arguments are required for its computation. An expresion is *quick* (as defined by Gray [6]), if it would take less time to evaluate an expression than to create another process to evaluate it. This is implementation-dependent, but a workable definition is to assume that all primitives are quick, and any combination of primitives is also quick. Quickness is determined relative to a *context*, which is the set of variables whose values are known to be available at that point. A function call is *immediately strict* if the portion of code it executes before it requires an argument it is strict with respect to is quick. This property is also determined relative to a context. Following are the formal definitions for these terms.

4.1.1. Strictness

A function is *strict* with respect to its *j*th argument if it requires the value of that argument for computation. For instance, cons is not strict with respect to either of its arguments, since it only needs to know where the values will be stored, and not what they are.

The COMMON LISP primitive `+` is strict with respect to all its arguments. It must have all the values to be able to return a result. The function `set` is strict with respect to only its first argument. It must know which symbol to assign the value to, but it need know only the address of the value.

4.1.2. Quickness

Gray [6] defines *quickness* in the following way:

Given a context <c>, an expression is *quick* in <c> if it is one of the following:

(1) a self-evaluating expression, such as a string or number

(2) quoted expression

(3) simple variable

(4) (f a_1 ... a_n) where f is either a primitive LISP operator or defined by

(defun f (v_1 ... v_n) E) or

(lambda (v_1 ... v_n) E)

where E is quick in <c> and for each a_j one of the following is true:

(a) if a_j is a variable, then either

(i) the value of a_j is available in <c>, or

(ii) E is not strict with respect to v_j.

(b) no parallel constructs appear in a_j (or its subexpressions) and a_j is quick in <c>, or

(c) f is not strict with respect to its *j*th argument (in other words, f is a constructor) and either

(i) a_j is quick in $<c>$, or

(ii) a_j is a parallel construct which does not require a wait.

(5) (if P E_1 E_2) where P, E_1, and E_2 are quick in $<c>$.

(6) (lambda (v_1 ... v_k) E)

As an example, (car x) is quick in the context <x> (meaning that the value of x is known to have been computed), but not in the contexts <>, <y>, etc. The primitive cons is quick in any context.

4.1.3. Immediate Strictness

A function is *immediately strict* with respect to its *j*th argument if it requires the value of that argument for computation, but the portion of the function code preceding the point of that requirement is *quick*. For example, the COMMON LISP primitive + may take several arguments, all of which are required to produce an answer. However, the hardware can add only two of the arguments at once, so there is a small delay before some of the arguments are needed. The + function is still considered immediately strict with respect to all arguments, because the computation done before these values are required is *quick*, since it involves only LISP primitives. An example of a function which is strict with respect to all its arguments, but not immediately strict, is the following:

```
(defun func1 (x y)
  (if (null x) (car y) (cdr y)))
```

This is immediately strict with respect to x, since its value is tested as the first action of the function. It is not immediately strict with respect to y, because it is not known how long the function will have to wait for the value of x if the value of x is being computed elsewhere.

4.2. Application of Concepts

Gray's basic idea of future placement is to place them only around the arguments to function calls, and only in the following cases:

(1) If the function is *immediately strict* with respect to its *i*th argument and there exists a later (further to the right) argument that is not *quick* in the current context, then enclose the *i*th argument in a future. This allows concurrent evaluation of required arguments.

(2) If the function is not *immediately strict* with respect to its *i*th argument, enclose it in a future. This allows evaluation of arguments concurrently with their use.

5. Description of SimPL

SimPL includes the same four constructs (future, delay, pcall, and touch) as Multilisp. It is implemented in COMMON LISP. and thus allows use of all its functions as well. Although SimPL is compiled, the user program must be run interpreted, for reasons explained in section 6.

5.1. Delay

The construct

```
(delay A)
```

returns a generated symbol whose value is a structure which contains all information necessary to evaluate the expression A at a later time. This symbol has `'delay` set to `t` on its property list. Evaluation of A will not begin until its value is actually required. Once this happens, the structure will be replaced by the value.

To guarantee consistent results between SimPL and a real multiprocessor system, if A is a function call, it should be a true function. That is, its value can be computed based solely on its arguments. Although closures save all variable bindings which are lexically visible at the time the closure is created, the values of these variables can be still be changed. Those local to the delay will not be affected by the rest of the program, but global variables may. The programmer should be aware of the possibility that the time at which a delay is evaluated could affect its result value.

5.2. Future

The construct

```
(future A)
```

causes the expression A to be evaluated, and outputs information about timing. It returns a generated symbol whose value is the result of the evaluation and which has `'future` set to `t` on its property list. The reason behind returning a symbol as a level of indirection is so that the simulator will be able to keep track of when a future's result is demanded. This indirection is not visible to the user program.

5.3. Touch

The construct

```
(touch A)
```

behaves as the identity operator, unless its argument is a future symbol, in which case it returns the future's result, or a delay symbol, in which case it computes the delay if undetermined, or returns its result if determined.

5.4. Pcall

The construct

```
(pcall F A B)
```

is implemented as

```
(apply (touch (future F)) (touch (future A)) (touch (future B)))
```

5.5. Implementation

The SimPL systems is intended to simulate the functioning of the above constructs on a multiprocessor system, but is actually run on a single processor. It outputs timing information about the "processes" created, from which the user may draw conclusions about the multiprocessing efficiency of the program simulated.

The system is implemented in three phases: the simulation, the analysis, and the output of plot data. Each phase requires complete output from the preceding one. Intermediate states are stored in files, and therefore the simulation need be run only once. Then the analysis and output phases may be repeated for each processor configuration desired.

5.5.1. Simulation Phase

The simulator takes an expression to be evaluated. It evaluates the expression sequentially, logging each occurrence of a future as if it were a separate process. It outputs the start and end time of each "process", plus any intermediate times when it is blocked or restarted. It also determines which processes would be started at the same time if an infinite number of processors were available. If the number of processors required is greater than 99, the program should be recompiled with a larger maximum number so as not to restrict the program's demand for additional processors.

Figure 1 shows a sample program, which is immediately strict with respect to both its arguments. Therefore, both of the arguments to this program should be computed as futures, since the program itself will block until all arguments are available. Figure 2 shows the output from the simulation phase when running the sample program. Package information and quoting of lowercase symbols has been edited out for clarity, and comments have been added. The first line is a comment containing the original expression to be evaluated. Following lines consist of an event number, a "process" name, an event type, and the cpu time used so far. All events with the same event number would be initiated at the same time if unlimited processors were available. A "-" in the event number field indicates that the timing of the event is directly dependent on some other event. Events are matched based on the "name" field. The names are composed of a tag (either "sim", "delay", or "future", depending on the type of process) concatenated with a unique number. The event types are: "begin-sim" and "end-sim", which represent the start and end of the simulation, "begin-exec" and "end-exec", which represent the start and end of the evaluation of a future, and "value-needed", which represents the point at which some process requires the value of the future named in the "id" field. The last lines are comments showing the final result of the simulation, the number of futures and delays created during execution, and the number of each whose values were actually required.

```
(defun sample (x y)
  (+ x y (* x y)))
```

Figure 1. Sample Program

```
(sample (future (sample (future (* 1 2))
                        (future (* 3 4))))
        (future (* 5 6)))

Event Name    Type          Time Comments

  0   sim1    begin-sim    18533 executing  entire  expres-
                                 sion
  1   future2 begin-exec   18616 (sample (future (* 1 2))),
                                 parent is sim1
  2   future3 begin-exec   18700 (* 1 2), parent is future2
  -   future3 end-exec     18766 future3 has completed
  2   future4 begin-exec   18816 (* 3 4), parent is future2
  -   future4 end-exec     18883 future4 has completed
  3   future3 value-needed 18916 future3's value needed by
                                 future2
  4   future4 value-needed 18950 future4's value needed by
                                 future2
  5   future3 value-needed 19000 future3's value needed by
                                 future2
  6   future4 value-needed 19033 future4's value needed by
                                 future2
  -   future2 end-exec     19100 future2 has completed
  1   future5 begin-exec   19150 (* 5 6), parent is sim1
  -   future5 end-exec     19183 future5 has completed
  7   future2 value-needed 19233 future2's value needed by
                                 sim1
  8   future5 value-needed 19266 future5's value needed by
                                 sim1
  9   future2 value-needed 19316 future2's value needed by
                                 sim1
 10   future5 value-needed 19366 future5's value needed by
                                 sim1
  -   sim1    end-sim      19416 the simulation has ended

; 1208
; 4 total futures, 0 futures computed but not needed
; 0 total delays, 0 delays not computed
```

Figure 2. Sample Output from Simulator

The events shown in Figure 2 are listed in the order they would occur in a single-processor environment, so that is the order in which they will be discussed.

The task named `sim1` is responsible for evaluating the entire expression

```
(sample (future (sample (future (* 1 2))
                        (future (* 3 4))))
        (future (* 5 6)))
```

It begins at time 18533. As it evaluates its arguments, it sees that the first one is

```
(future (sample (future (* 1 2)) (future (* 3 4))))
```

which is a future, so it creates a new task named `future2` to execute

```
(sample (future (* 1 2)) (future (* 3 4)))
```

This task is assigned event number 1, since it is the first such task to be created. It begins at 18616.

This new task now evaluates its arguments, discovers that the first one is a future, and creates another new task named `future3`, which begins at 18700, to evaluate

```
(* 1 2).
```

This task receives an event number of 2, since it cannot possibly occur at the same time as the task with event number 1, which is its parent task. It finishes at 18766. The second argument is also a future, and a new task, `future4`, is created to evaluate

```
(* 3 4)
```

It finishes at 18883. Next, at 18916, the value of `future3` is required by its parent task, `future2`. At 18950, the value of `future4` is also required. The two values are required again respectively at 19000 and 19033. Then `future2` finishes, at 19100.

The original process, `sim1`, is now ready to evaluate its second argument,

```
(future (* 5 6)).
```

This is also assigned event number 1, since it occurs at the same time as the first argument (there is no intervening computation between the creation of the future for the first argument and this one). It begins at 19150 and finishes at 19183. Finally, the parent task `sim1` requires the values of its two arguments, `future2` and `future5`, at 19233 and 19266, and again at 19316 and 19366. It finishes at 19416.

The final result is 1208. Four futures were created during computation, and all their values were eventually required. No delays were created.

5.5.2. Analysis Phase

This program reads data generated by the simulation phase, with the number of processors available for the simulation being specified as an argument. It constructs a tree of processes, and outputs timing data for each event relative to its parent process. It also runs the output phase.

5.5.2.1. Bounded Analysis

Figure 3 shows some sample output from the analysis phase. The first three fields are the same as before, but with the addition of some new types: "value-received", which represents the point at which a process actually receives the value of a future, "blocked", which represents the time at which a process is blocked, either because it is waiting for a future to be determined or because there are not enough processors and it has been pushed onto a queue to await processing, and "restarted", which represents the time such a process is able to continue. The cpu time field is now relative to the start time of the process in which that event occurred. For instance, on line 2 of Figure 3, `future2` starts 83 time units after its parent, `sim1`, started. The final field is a sequencing number to keep events which occur at the same time in the proper order for the output phase.

```
Event  Name        Type         Time Sequence

 0    sim1    begin-sim          0        1
 1    future2 begin-exec        83        2
 1    future5 begin-exec        83        3
 -    future5 end-exec         116        4
 2    future3 begin-exec       167        5
 2    future4 begin-exec       167        6
 7    future2 value-needed     183        7
 -    sim1    blocked          183        8
 -    future3 end-exec         233        9
 -    future4 end-exec         234       10
 3    future3 value-needed     250       11
 -    future3 value-received   250       12
 4    future4 value-needed     284       13
 -    future4 value-received   284       14
 5    future3 value-needed     334       15
 -    future3 value-received   334       16
 6    future4 value-needed     367       17
 -    future4 value-received   367       18
 -    future2 end-exec         434       19
 -    sim1    restarted        434       20
 -    future2 value-received   434       21
 8    future5 value-needed     467       22
 -    future5 value-received   467       23
 9    future2 value-needed     517       24
 -    future2 value-received   517       25
10    future5 value-needed     567       26
 -    future5 value-received   567       27
 -    sim1    end-sim          617       28
```

Figure 3. Sample Output from Analyzer

5.5.2.2. Unbounded Analysis

Unbounded analysis is currently done by preallocating "enough" processors so that no process ever blocks for lack of a processor. For the simulations done in this paper, 99 was enough.

5.5.3. Output Phase

The third stage reads the previous results and produces plotting data. Figure 4 shows some sample output from the plot output phase. Each line consists of an x value (cpu time) and a y value (number of processors in use). At the end of the plotting output, a comment section gives the maximum number of processors that were in use at one time, the grand total of processor time used over all processors, the "reserved" processor time (the maximum number of processors multiplied by the longest execution thread), and the efficiency of the system (the percentage of the reserved time which was actually used). Figure 5 shows a

```
Time     Processors in Use

   0     0
   0     1
  83     1
  83     3
 116     3
 116     2
 167     2
 167     4
 183     4
 183     3
 233     3
 233     2
 234     2
 234     1
 434     1
 434     1
 617     1
 617     0
; Maximum of 4 processors in use
; Processor time used:  883
; Total processor time available:  2468
; Percentage used·  35.78
```

Figure 4. Sample Output from Plot Generator

sample plot.

6. Details of Implementation

The following sections give an outline of how SimPL works, some design decisions which were made along the way, and some problems with the current implementation.

6.1. Program Outline

Here is a description of the three phases of SimPL.

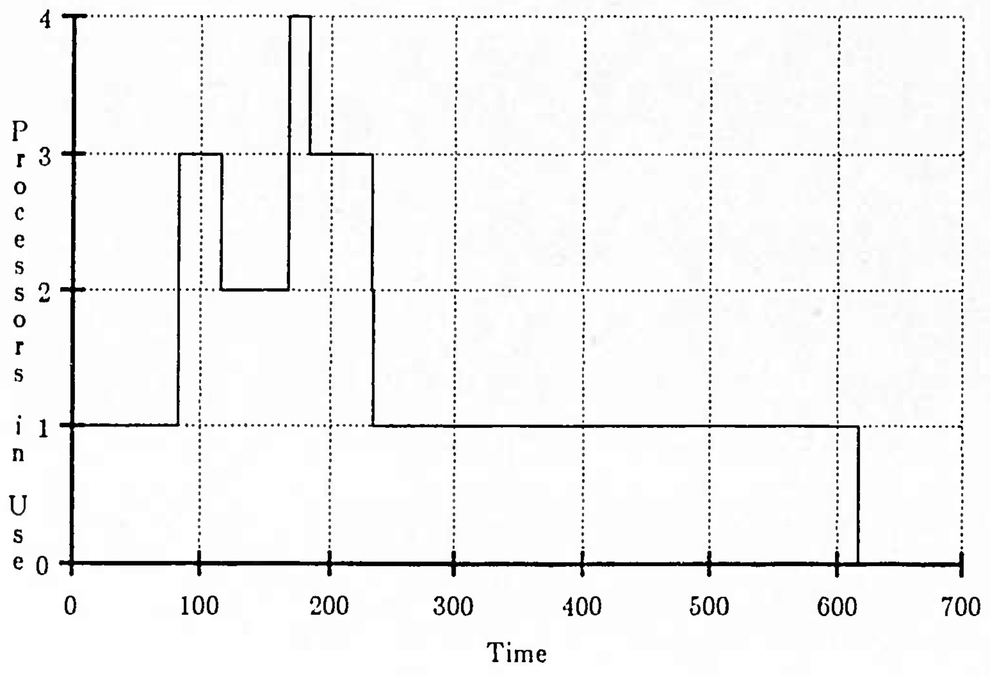

Figure 5. Sample Plot

6.1.1. Simulator

The LISP simulator evaluates the form given it, removes all indirection resulting from future and delay structures, and then returns and prints this "cleaned-up" result.

Since futures and delays look like function calls in the program, one might think that they could actually be implemented that way. Normal LISP evaluation cannot be used; if `future` were a function call, its argument expression would be evaluated before the the future function were entered, and all chance to time the expression is gone. For example, in the expression

```
(future (+ 1 2))
```

the `future` function would receive the integer `3` rather than `(+ 1 2)`. Treating delays as function calls does not work either. For example, delays can be used to implement streams:

```
(defun integers (i)
   (cons i (delay (integers (+ i 1)))))
```

This will return a stream of all integers starting with `i`, but will only calculate the ones which are needed. In this case, fully evaluating the argument to a delay would lead to a non-terminating calculation.

Another problem is that when a future or delay is begun, it returns a symbol that will eventually point to the result. This level of indirection remains throughout the simulation, and thus whenever an object is evaluated it must be checked to see if it is a future or delay. The user program should not be affected by this indirection.

Both of these problems are solved by taking control of the evaluation process away from the LISP evaluator. In COMMON LISP it is possible for the user-written program to gain control of the evaluation process through two hooks in the interpreter: `evalhook`, which controls execution of interpreted code, and `applyhook`, which controls execution of compiled code.

When eval is called on a form and the value of the symbol *evalhook* is non-nil, then *evalhook* and *applyhook* are temporarily set to nil, and control is passed to the function named by the former value of *evalhook*. This function is responsible for evaluating the form and returning its value, and will usually call evalhook to do this.

Applyhook works in much the same way, except that the function named by its value will receive a function and a set of already evaluated arguments. However, since applyhook operates on compiled code, it will (depending on the implementation) not be called to evaluate certain special forms, such as car and cdr. This means that, in *compiled* code, there is no way to check the argument to one of these functions to see if it is a future. Therefore, the programs run under the simulator must be interpreted, resulting in considerable loss of speed. The simulator itself can be compiled, and is.

The simulator decides what processes would be started at the same time if infinite processors were available, and assigns event numbers to these processes accordingly. Each future or delay at the same level in an expression is assigned the same event number, unless there is an intervening normal expression, in which case the event number must be incremented. It is also incremented whenever the first future at a level is encountered. For instance, in the expression

```
(+ (future A) (future B) (* 1 2) (future C))
```

the first two futures would be considered to have started at essentially the same time, but the third would start later because of the intervening multiplication. This is done by keeping track of the depth inside an expression, and the type (future or non-future) of the last expression evaluated at that depth.

During evaluation of the arguments to a function call, if a future or delay is encountered, it is necessary to know whether the function can proceed without its value (whether the function is *immediately strict*). The simulator knows which COMMON LISP built-in functions are immediately strict only through the fact that this information has been hard-coded into the simulator itself. At load time, a series of commands are executed which add the atom *strict* to the property list of the name of each strict function. Functions which are not assigned this property are assumed to be non-strict. This causes the program to take longer to load, but it need be done only once and speeds program execution greatly by making the test for strictness (which must be done every time a function is called) fast. The choice was made to indicate strictness rather than nonstrictness, which results in all user functions being nonstrict by default. This is correct, since they are composed of LISP functions, which may or may not be strict themselves.

6.1.2. Analyzer

The analyzer works by building a tree of "process" structures. Each structure contains information about the process start and end time, the amount of time it has spent in a blocked state, its child processes, etc. An array of "processor" structures is also created. The analysis is begun by placing the simulation "process" into a "processor". Then the next event in the life of each process which is currently "running" is collected, and the one which would occur first is processed. This could involve creation of a new process or blockage or ending of an old one. The next events are collected and processed again until all are done.

6.1.3. Output

The output phase indicates the efficiency of the simulated system. It sums the processor time used by all the different processes and divides by the total reserved processor time -- the time taken by the longest thread of execution multiplied by the number of processors in the system. If the analysis is for an unbounded number of processors, then the maximum number of processors actually used at least once is taken to be the number reserved. For a single processor, efficiency will always be 100 percent, since the processor is constantly in use throughout execution. For multiple processors, some may be used for only a very short time. Such a processor could be left idle for the whole run-time by suffering a small delay while waiting on an already busy one. The cost of using a full processor to avoid this delay may not be realistic if many independent job streams are available to keep processors busy (as in a time-shared environment).

6.2. Design Decisions

6.2.1. Methods for Simulation

Two methods were considered:

(1) Simulate multiprocessors by spawning subprocesses. A shared program or file would be needed to keep track of the process queue and the number of idle processors.

(2) Execute each "process" sequentially, keeping track of the start and stop times.

The first method has the advantage of being easier to implement and observe, but the overhead of forking new LISP processes on UNIX and the requirement of sufficient swap space is too high. Also, the requirement for shared memory would probably necessitate the use of a disk file and a locking mechanism, which would further slow things down. Lastly, there is a problem of how to return a LISP object created in a child process to the parent.

The second method has the advantage of speed, smaller execution size, and simplicity. The disadvantage lies in trying to keep track of which processes would be running concurrently on a multiprocessor. However, the advantages of this method seem to outweigh this problem, so this is the method chosen.

6.2.2. Scheduling of Futures

Several methods were considered for scheduling the evaluation of futures:

(1) Immediate evaluation. When a future is encountered, evaluate it right away and store its value. Log all pertinent information about timing. This method has the advantage of being the simplest to implement.

(2) Delayed evaluation. When a future is encountered, treat it as a delay. When its value is required, evaluate it and log all pertinent information about timing. This method has the advantage of not having to compute futures whose values are never needed. However, this does not fit with the definition of Multilisp futures, which says that their computation begins as soon as there is an available processor, whether the value is required or not.

(3) Stack evaluation. Push each future on a stack in order, and when one is needed, compute all older (or newer) ones before it. For instance,

```
(F (future A) (future B) (future C) (future D))
```

would push A, B, C, and D onto the stack, and if the value of C was the first one required, it would trigger the evaluation of A and B first. Future D would not be triggered. This has the problem that futures which are not yet needed may be forced to finish being calculated, thus using more resources at one time than is necessary.

I chose the method of immediate evaluation, since it is consistent with Multilisp, and also provides futures as a distinct construct from delays.

6.3. Problems

Following is a discussion of a few problems with SimPL.

6.3.1. Extra Strictness

In SimPL, a function must be declared as strict with respect to all or none of its arguments. Unfortunately, the definition of strictness applies separately to each argument to the function. Thus, a function which is strict with respect to only some of its several arguments must be declared strict, even though this will give erroneous results in some cases.

6.3.2. Extra Indirection

In Multilisp, the structure for a determined future is replaced by the future's value whenever a garbage collection is done. This was not done in SimPL, so as to be able to determine which objects are values of futures, and therefore be able to keep track of references to them. Also, in some implementations of LISP, this is not possible, since the type of an object is determined from its address in memory. Therefore this level of indirection must be left in the program.

6.3.3. Memory Usage

The simulator turns off garbage collection (under those implementations of COMMON LISP where this is possible) so as to get more consistent times. Even if it were not turned off, the values of futures and delays are kept throughout the program, whether they are needed or not, because these values are assigned to symbols interned in the "simpl" package. Therefore, it is possible that a large simulation with many futures could run out of memory. The user should take whatever measures are necessary in the particular implementation of COMMON LISP to insure that enough dynamic memory is allocated prior to running the simulation.

6.3.4. Side Effects

Multilisp handles side effects by simply ignoring them. It is up to the programmer to insure that either there are no side effects, or that they will not adversely affect the results of the program. SimPL also ignores side effects, but programs which use them will perhaps show a different result than if executed on a multiprocessor, because the simulator actually runs the program sequentially.

6.3.5. Incomplete Delays

The arguments to delays should be true functions, since the values of global variables could change in the time between creation of the delay and execution of it.

7. Execution of SimPL

This section includes instructions for the use of SimPL, and a description of the programs which were analyzed for this project.

7.1. Instructions for Use

To prepare for simulation, change to the directory ~penn/simpl on ucbarpa and execute the following commands at the LISP interpreter:

```
(load "main.lisp")
(use-package 'simpl)
(load "your-program")
```

To run the simulator, enter an expression of the form

```
(sim expr [outfile])
```

to the LISP interpreter, where `expr` is an expression which calls the program to be simulated, and `outfile` is the optional file to send the results to (this must be specified if the results are to be analyzed). For example:

```
(sim (qsort '(3 1 2)))
```

or

```
(sim (qsort '(3 1 2)) results.dat)
```

Next, the simulator output is analyzed:

```
(analyze infile [outfile [nprocs]])
```

`Infile` should be the name of a file generated with the `sim` function, and `outfile` is the name of an optional output file. If it is not specified, output goes to the terminal. `Nprocs` is the number of processors to use in the simulation. If it is not specified, the simulation will use as many processors as needed (up to 99). The output of this command is a set of x-y pairs with commented labels, suitable for input to a simple plotting program. When plotted, these show the number of processors active at a given time.

7.2. Analysis of Programs

Futures were manually inserted into the following programs (except QSORT) according to the method given in Gray's paper [6]. They were then run in an unbounded environment to determine the maximum number of processors which could possibly be required. Then, they were run in bounded environments with varying numbers of processors between 1 and the maximum. Data from these runs were used to produce the following graphs.

All programs analyzed in this section are included in Appendix A.

7.2.1. TAK

TAK is one of the Gabriel benchmarks [4], useful mainly for testing the speed of function calls. It is used here because of its potential for exponential parallelism.

There are three futures in the program. Since TAK is immediately strict with respect to its first two arguments, and they are followed by a non-strict, non-quick argument, futures must be placed around each of the fiist two. Since TAK is *not* immediately strict with respect to its last argument, the last also should be enclosed in a future.

Figure 6 shows the results of the analysis on a call to `(tak 9 6 3)`. The X axis shows the progress of time in simulation units; the Y axis shows the number of processors which were busy during each time period.

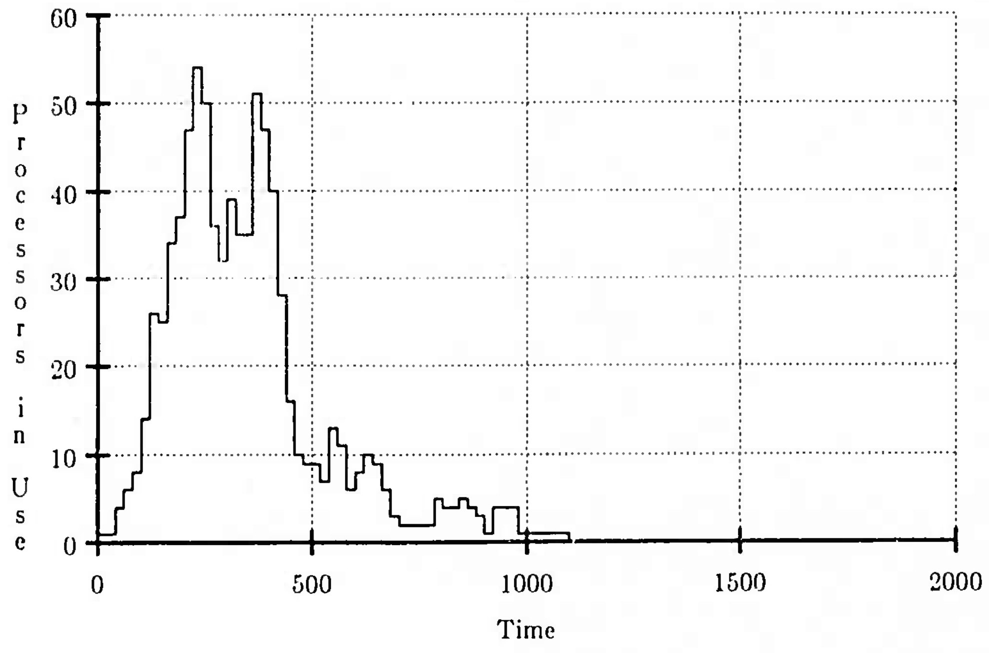

Figure 6. TAK: Number of processors in use

The following figures show the results of running four different invocations of TAK: `(tak 6 4 2)`, `(tak 9 6 3)`, `(tak 4 2 1)`, and `(tak 7 5 2)` (marked by squares, circles, triangles, and diamonds, respectively). Figure 7 shows how the run time changes with the number of processors in the system. Note that it drops quite sharply over the addition of the first 4 processors, and then levels out.

Figure 8 shows how the efficiency of the system changes with the number of processors available. Jaggedness in the lines probably represents inconsistencies in the time statistics gathered from running programs on a real system. The efficiency drops steadily as processors increase, but at different rates depending on the problem.

Figure 9 shows how the speedup changes with the number of processors available. The speedup is computed by divided the sequential run time (with futures) by the parallel run

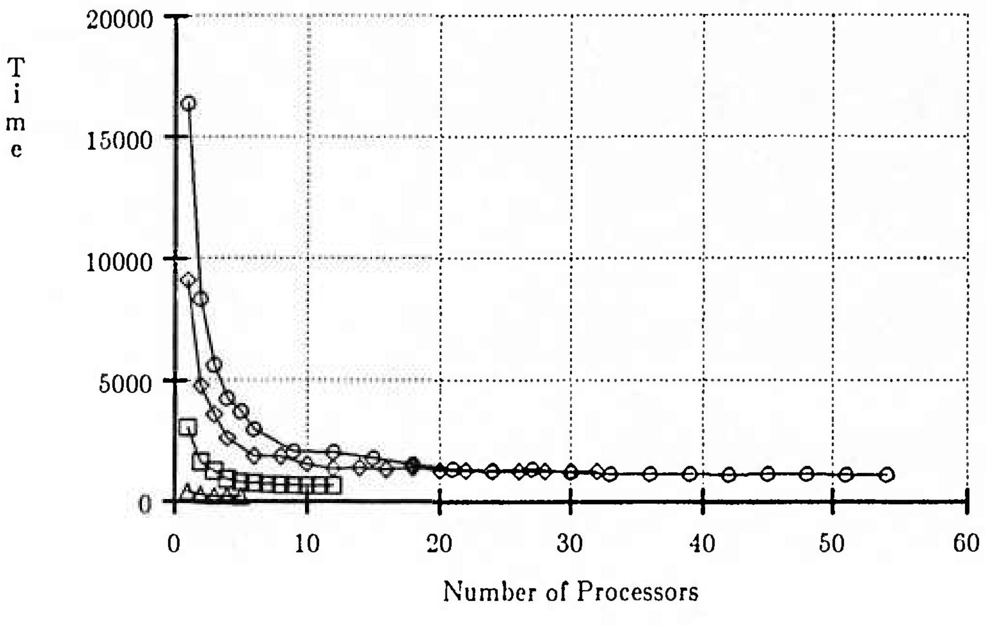

Figure 7. TAK: Run Time Vs. Number of Processors

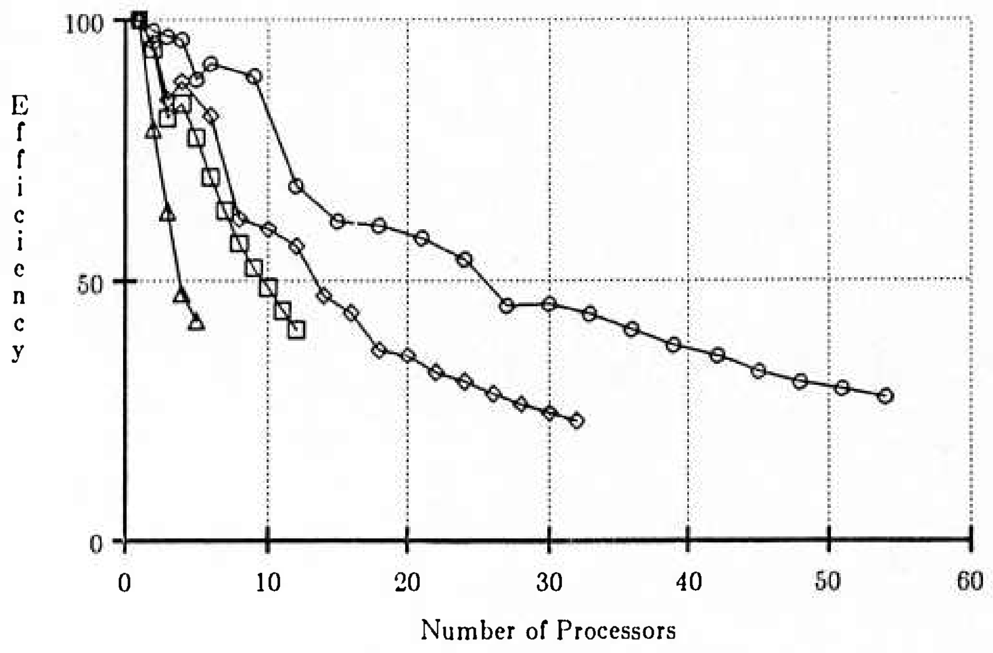

Figure 8. TAK: Efficiency Vs. Number of Processors

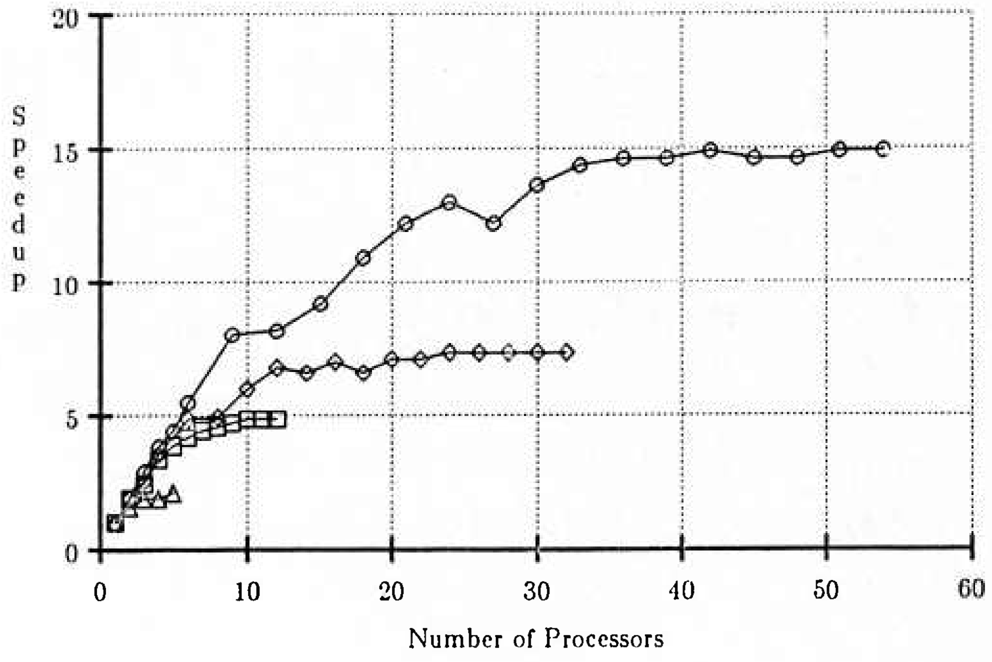

Figure 9. TAK: Speedup Vs. Number of Processors

time. This also levels out quickly, but after a higher number of processors are added, again varying with the problem.

7.2.2. QSORT

QSORT is an implementation of Quicksort, complete with futures, taken from Halstead's paper [8]. The program contains six futures.

Figure 10 shows the results of the analysis on a quicksort of a list of 20 randomly selected integers.

As with TAK, the following figures show four different invocations of QSORT, on lists of 20, 30, 40, and 50 randomly generated integers (marked by squares, circles, triangles, and diamonds, respectively). Figure 11 shows how the run time changes with the number of processors available.

Figure 12 shows how the efficiency of the system changes with the number of processors available. Figure 13 shows how the speedup changes with the number of processors available. All three of these graphs show much more consistency across different levels of complexity. QSORT is by far the most efficient of the three programs studied here.

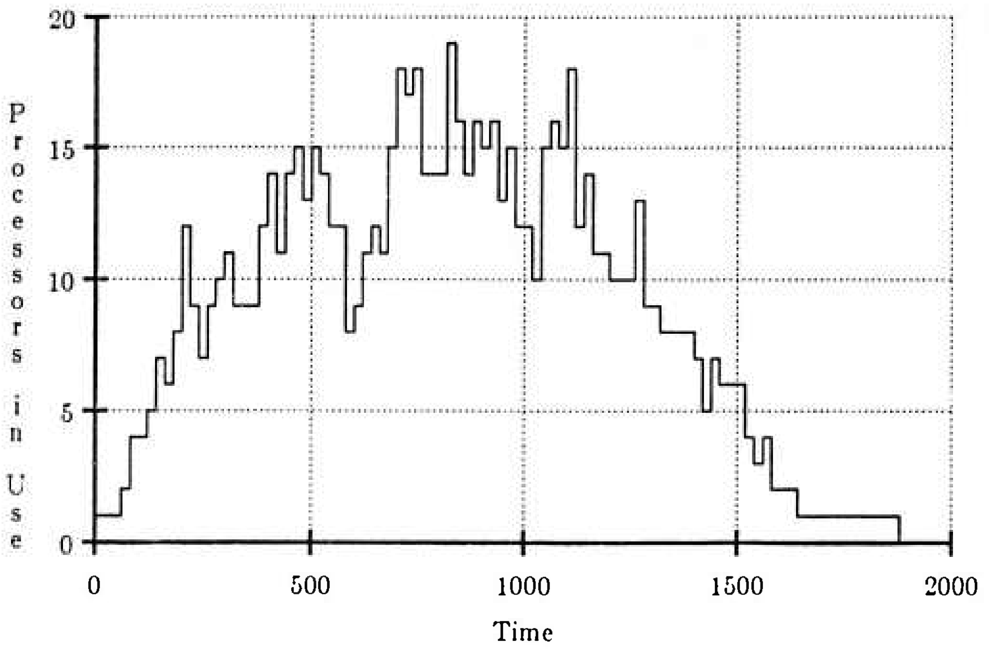

Figure 10. QSORT: Number of processors in use

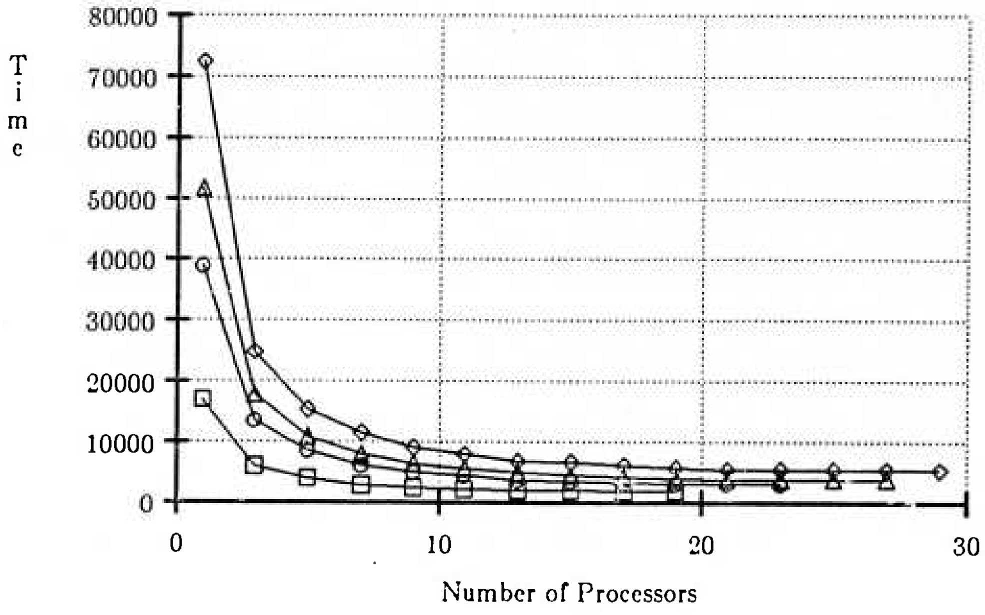

Figure 11. QSORT: Run Time Vs. Number of Processors

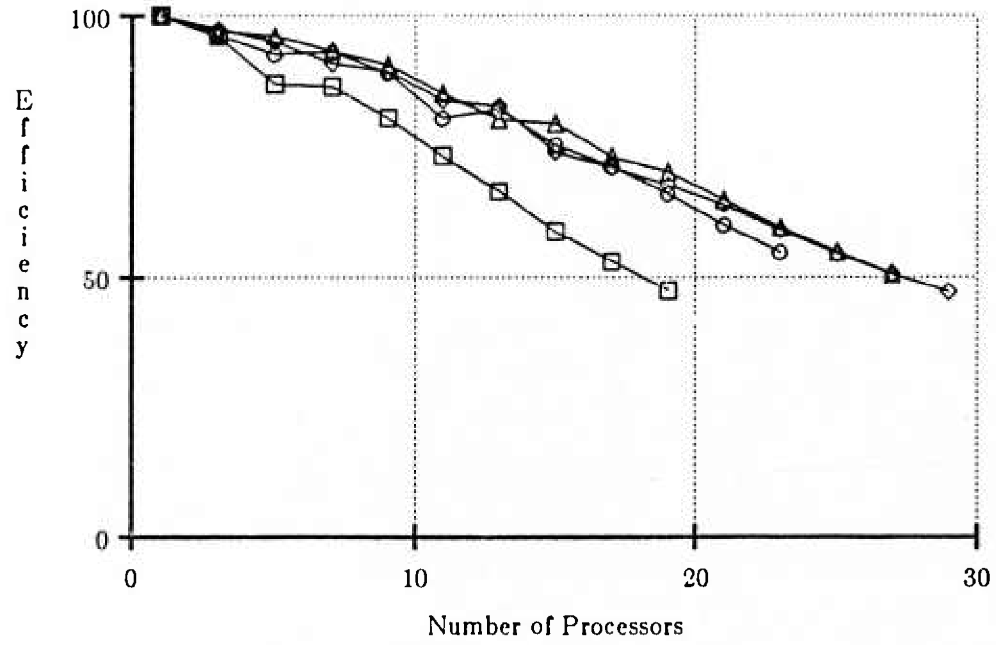

Figure 12. QSORT: Efficiency Vs. Number of Processors

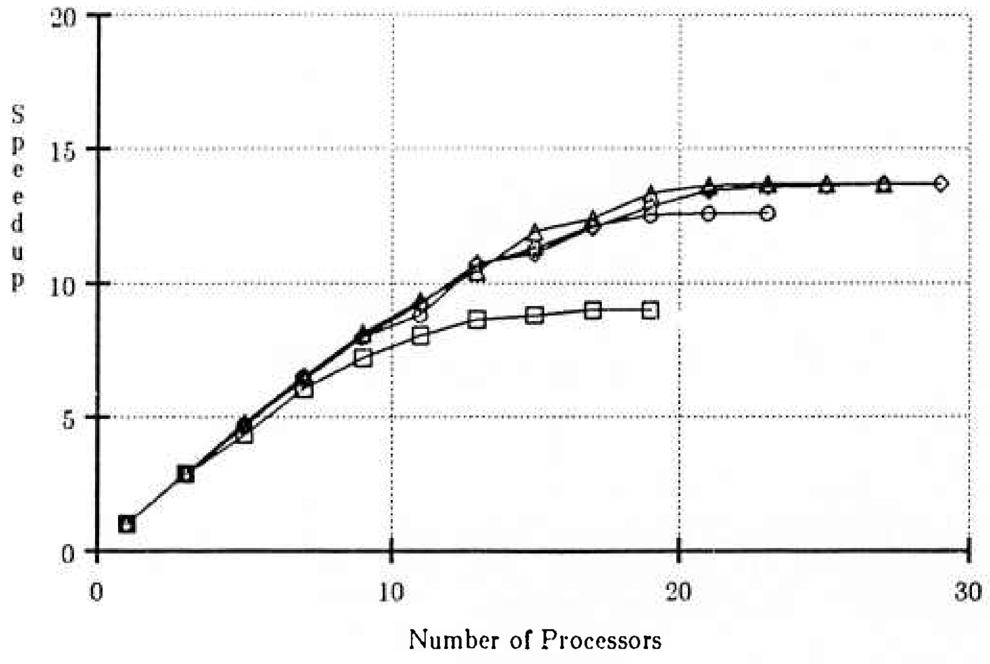

Figure 13. QSORT: Speedup Vs. Number of Processors

7.2.3. FRPOLY

Futures were manually inserted in FRPOLY according to the method given in Gray's paper [6]. Each function has been categorized according to its strictness or immediate strictness with respect to each of its arguments. This categorization is given in the comments before each function. Fourteen futures have been added to FRPOLY.

Figure 14 shows the results of the analysis on FRPOLY. The three different invocations of FRPOLY are: $(x+y+z+1)^2$, $(x+y+z+1)^3$, and $(x+y+z+1)^4$ (marked by squares, circles, and triangles, respectively). The expression (x+y+z+1) has already been computed before the simulation starts.

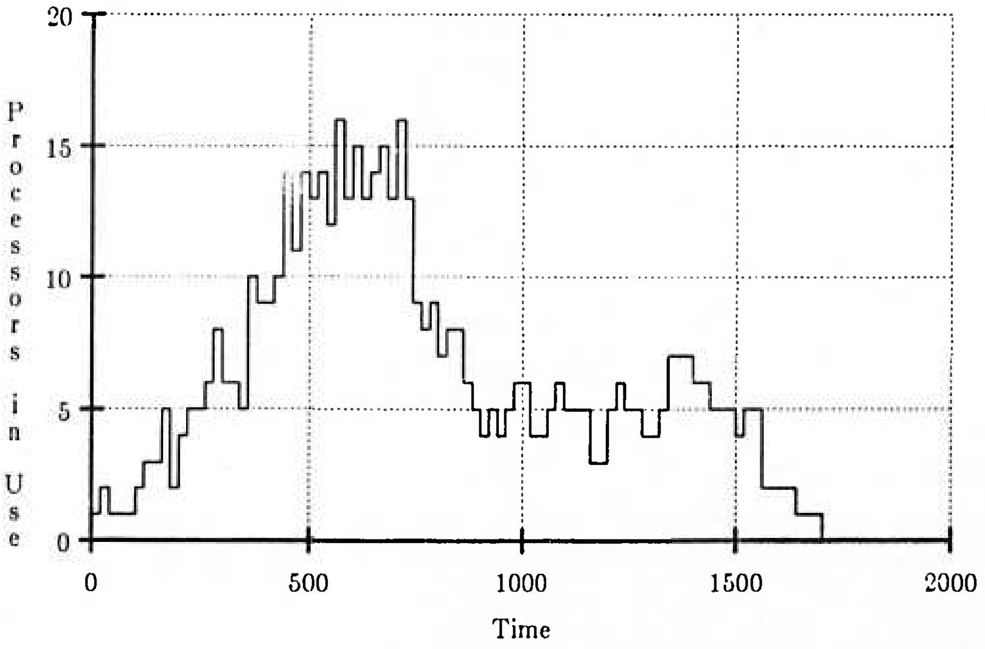

Figure 14. FRPOLY: Number of processors in use

Figure 15 shows how the run time changes with the number of processors available. Figure 16 shows how the efficiency of the system changes with the number of processors available. Figure 17 shows how the speedup changes with the number of processors available. FRPOLY seems similar to TAK in its lack of efficiency and inconsistency. It may well be that the test runs were not complex enough to show consistency, as suggested by the fact that the higher-complexity runs of QSORT tended to bunch together.

7.3. Results of Tests

Usually the speedup of a parallel program is calculated by dividing the sequential run time by the parallel run time. In the best of all possible worlds, this would be equal to N, the number of processors in the parallel system. If colluding processes are used, the speedup could even be greater than N.

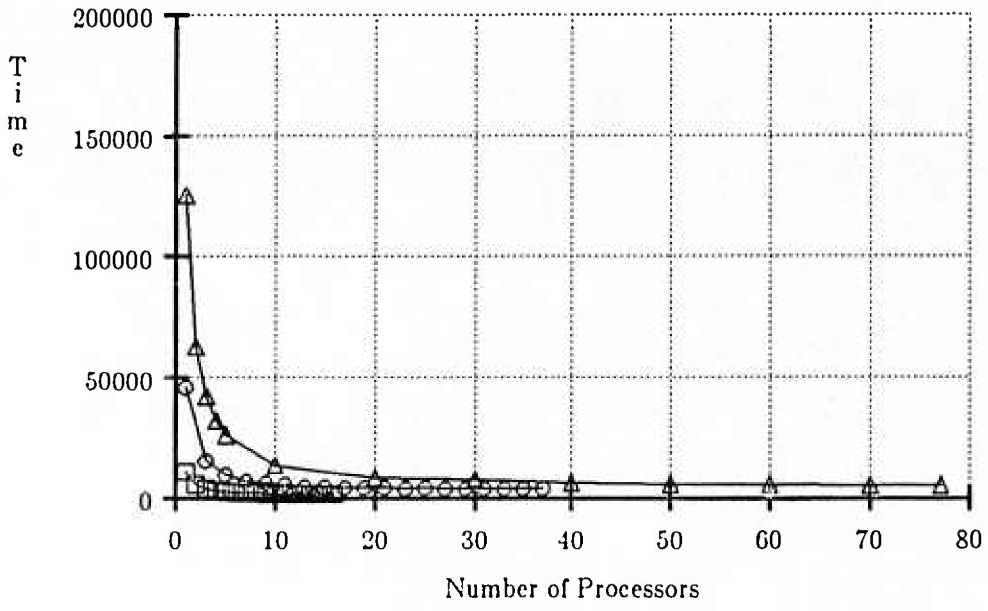

Figure 15. FRPOLY: Run Time Vs. Number of Processors

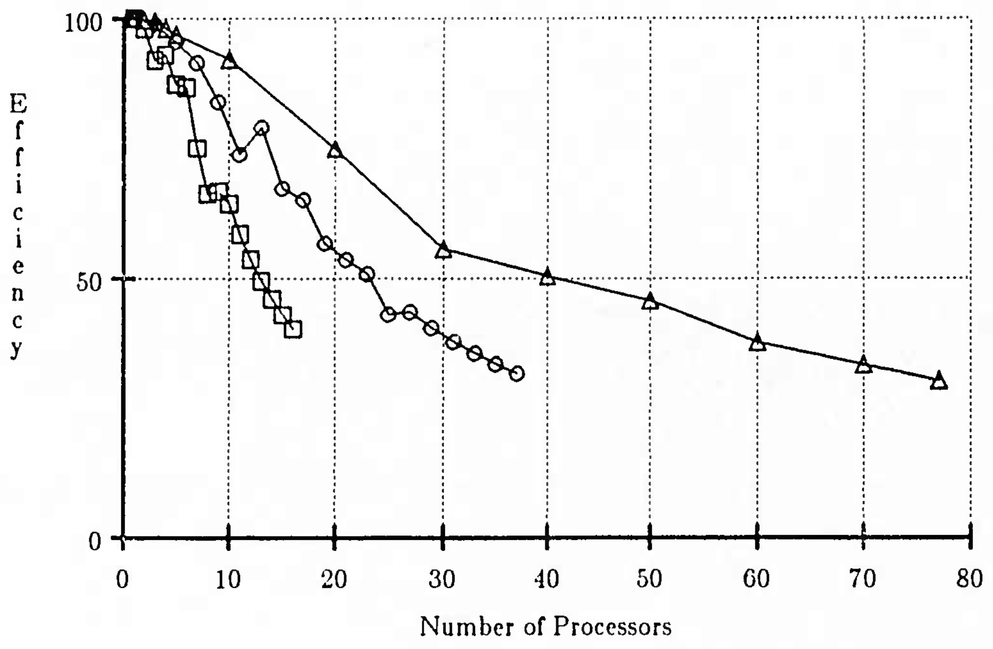

Figure 16. FRPOLY: Efficiency Vs. Number of Processors

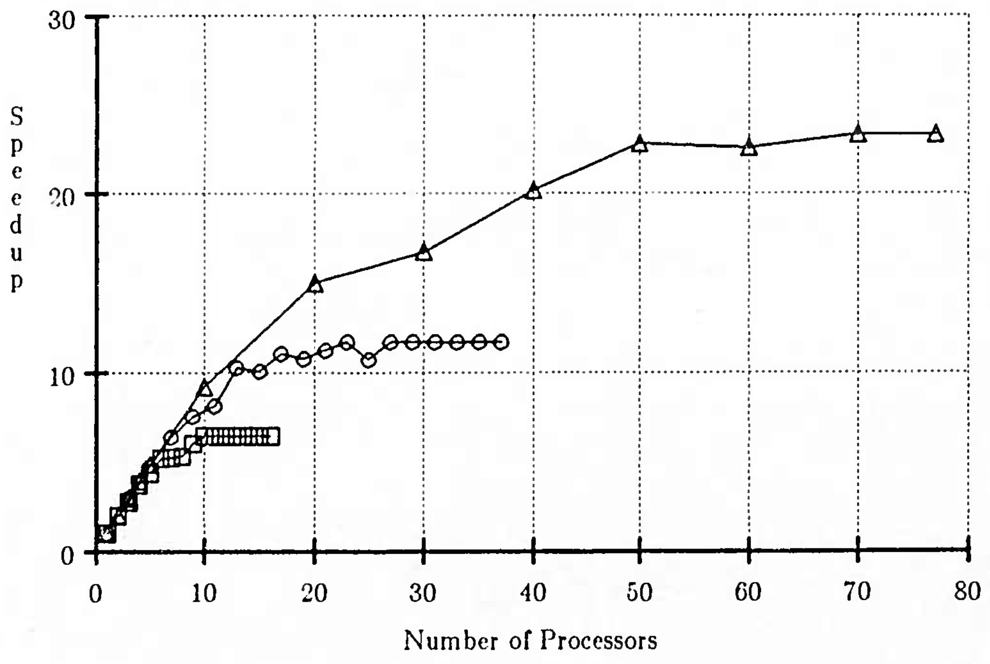

Figure 17. FRPOLY: Speedup Vs. Number of Processors

Comparing the run time of a program in a multiprocessing environment with the run time of the same program in a uniprocessing environment certainly is useful in describing performance gain. However, it does not give any information about why the speedup occurred: some speedup was gained through parallel computation, and some lost through scheduling and process creation overhead. In order to get this information, one must do two comparisons:

(1) The run time in a uniprocessing environment against the run time in a multiprocessing environment which happens to have only one processor.

(2) The run time in a multiprocessing environment with one processor against that in an environment with N processors. The first comparison gives an approximation of the

scheduling overhead. It will probably be too high, since naturally processes will block more often with fewer processors. The second gives the speedup due to use of parallel computation.

Figures 18, 19, and 20 show the overhead comparisons. The first line of data shows the run times for various tests in the uniprocessor environment, i. e., the programs were run under SimPL without any futures added. The second line shows the run times for the same programs, but with futures added. The simulator is limited to one processor. A comparison of these two lines shows the overhead associated with adding futures to a program, without any parallel processing. The third line shows the run times for the programs with futures on an unbounded system. These numbers also happen to be the endpoints of the speedup graphs

Environment		Program: tak		
# Futures	# Processors	6 4 2	9 6 3	4 2 1
0	1	740	3300	100
3	1	3020	16380	380
3	∞	620	1100	180

Figure 18. TAK: Comparisons Showing Overhead

Environment		Program: qsort		
# Futures	# Processors	20 items	30 items	40 items
0	1	2540	5420	6920
6	1	16920	39040	51660
6	∞	1880	3100	3780

Figure 19. QSORT: Comparisons Showing Overhead

Environment		Program: frpoly		
# Futures	# Processors	$(x+y+z+1)^2$	$(x+y+z+1)^3$	$(x+y+z+1)^4$
0	1	4000	14080	31860
14	1	10940	45900	125420
14	∞	1700	3920	5380

Figure 20. FRPOLY: Comparisons Showing Overhead

shown previously.

The overhead or futures is quite large, but the programs still show speedup in spite of this.

8. Conclusions

8.1. Directions for Further Work

The overhead involved in scheduling processes, creating futures and delays, and checking their values is not monitored by SimPL. The insertion of more calls to a timing function in the evaluation hook function is necessary -- the time should be recorded every time the simulator changes from executing SimPL code to user code.

The interaction between processes using the same global data structures is not simulated, since the simulation is really run sequentially. "Locks" could probably be implemented in much the same way as other events in the system.

Functions must be declared strict with respect to all or none of their arguments; no allowance is made for functions which are strict with respect to only some of their arguments. This can cause the value of a future or delay to be demanded prematurely if it is passed as an argument to a partially strict function. A simple fix would be to change the value of the atom *strict* on the function name's property list to be a list of numbers indicating the arguments it is strict with respect to.

Multiple values should be handled. Currently the hook function works only on functions which return single values. All but the first of multiple values will be discarded.

This system would be greatly improved if it were made a part of the LISP system rather than riding atop it. It would run faster, and the indirection in accessing the values of futures and delays could be eliminated after they had finished evaluation.

The 99 processor limit is imposed because the "processors" are stored in an array. This limit could be eliminated by using COMMON LISP adjustable arrays, or even a list, to store the processors. When all were busy, and another was requested, it could be easily created and added to the system.

8.2. Summary of Results

The programs run for this project each use a different mix of symbolic and numerical computing. However, the results for all indicate that the possibilities for speedup with parallel execution are indeed limited, but vary with the problem under consideration. The leveling off of both speedup and efficiency in larger systems shows that the maximum benefit can be achieved without using excessively large systems. The use of simulation to determine this maximum before purchasing such a system would be wise.

APPENDIX A

Sample Programs

A.1. TAK

```
;;; Based on TAK in
;;;    Gabriel, "Performance and Evaluation of Lisp Systems" [4]

(use-package 'simpl)

; Immediately strict in x and y.
(defun tak (x y z)
  (if (not (< y x))
      z
      (tak (future (tak (1- x) y z))
           (future (tak (1- y) z x))
           (future (tak (1- z) x y)))))
```

A.2. QSORT

```
;;; Based on Halstead's qsort in
;;;    Halstead, "Multilisp:  A Language for Concurrent
;;;         Symbolic Computation" [8]

(use-package 'simpl)

(defun qsort (l)
  (qs l nil))

(defun qs (l rest)
  (if (null l)
      rest
      (let ((parts (partition (car l) (cdr l))))
      (qs (car parts)
          (future (cons (car l) (qs (cdr parts) rest)))))))

(defun partition (elt lst)
  (if (null lst)
      (cons nil nil)
      (let ((cdrparts (future (partition elt (cdr lst)))))
      (if (> elt (car lst))
          (cons (cons (car lst)
                      (future (car cdrparts)))
```

```
                    (future (cdr cdrparts)))
                (cons (future (car cdrparts))
                    (cons (car lst)
                        (future (cdr cdrparts))))))))
```

A.3. FRPOLY

```
;;; Based on Fateman's FRPOLY in
;;;    Gabriel, "Performance and Evaluation of Lisp Systems" [4]

(use-package 'simpl)

;;; Checks to see if its argument is zero.
;;;   pzerop is quick in <x>, and immediately strict
;;;   with respect to x.
(defmacro pzerop (x)
  '(let ((x ,x))
     (if (numberp x)
       (zerop x))))      ;true for 0 or 0.0

;;; polynomial representation looks like (var (exp . coef) (exp . coef) ...)
;;; where the exp's [exponents] are in descending order, no zero coefs are
;;; allowed, and a form (x (0 . 5)) [i.e. 5*x^0, or 5] is reduced to 5.
;;; Coefs can be (recursively) polynomials in other variables.

;;; Adds an exponent-coefficient pair to a coefficient list x.
;;;   pcoefadd is immediately strict with respect to its
;;;   second argument, c, but not strict with respect to e
;;;   and x.  It is quick in <c>.
(defun pcoefadd (e c x)
  (if (pzerop c)
      x
      (cons (cons e c) x)))

;;; Add a constant c to a polynomial p.
;;;   pcplus is immediately strict with respect to p,
;;;   and strict with respect to c.  It is not quick.
(defun pcplus (c p)
  (if (atom p)
      (+ p c)
      (psimp (car p) (pcplus1 c (cdr p)))))

;;; Adds a constant c to a coefficient list x.
;;;   pcplus1 is immediately strict with respect to x,
;;;   and strict with respect to c.  It is not quick.
(defun pcplus1 (c x)
  (cond ((null x)
         (if (pzerop c)
             nil
```

```
              (cons (cons 0 c) nil)))
         ((pzerop (caar x))
          (pcoefadd 0 (pplus c (cdar x)) nil))
         (t
          (cons (cons (caar x) (cdar x)) (future (pcplus1 c (cdr x)))))))

;;; Multiply a constant c by a polynomial p.
;;;    pctimes is immediately strict with respect to p,
;;;    but not strict with respect to c.  It is not quick.
(defun pctimes (c p)
  (if (atom p)
      (* c p)
      (psimp (car p) (pctimes1 c (cdr p)))))

;;; Multiply a constant c by a coefficient list x.
;;;    pctimes1 is immediately strict with respect to x,
;;;    but not strict with respect to c.  It is not quick.
(defun pctimes1 (c x)
  (if (null x)
      nil
      (pcoefadd (future (caar x))
             (future (ptimes c (cdar x)))
             (future (pctimes1 c (cdr x))))))

;;; Add two polynomials x and y
;;;    pplus is immediately strict with respect to x,
;;;    and strict with respect to y.  It is not quick.
(defun pplus (x y)
  (cond ((atom x) (pcplus x y)) ; is x a constant?
        ((atom y) (pcplus y x)) ; is y a constant?
        ((eq (car x) (car y))
         (psimp (car x) (pplus1 (cdr y) (cdr x))))
        ((> (get (car x) 'order) (get (car y) 'order))
         (psimp (car x) (pcplus1 y (cdr x))))
        (t (psimp (car y) (pcplus1 x (cdr y))))))

;;; Add two coefficient lists
;;;    pplus1 is immediately strict with respect to x, but
;;;    not strict with respect to y.  It is not quick.
(defun pplus1 (x y)
  (cond ((null x) y)
        ((null y) x)
        ((= (caar x) (caar y))
         (pcoefadd (caar x)
                (future (pplus (cdar x) (cdar y)))
                (future (pplus1 (cdr x) (cdr y)))))
        ((> (caar x) (caar y))
         (cons (cons (caar x) (cdar x)) (future (pplus1 (cdr x) y))))
        (t (cons (cons (caar y) (cdar y)) (future (pplus1 x (cdr y)))))))

;;; Simplify a polynomial.
;;;   psimp is immediately strict with respect to x, but
;;;   not strict with respect to var.  It is quick in
;;;   <x, (car x), (caar x)>
(defun psimp (var x)
  (cond ((null x) 0)
        ((atom x) x)
        ((pzerop (caar x)) (cdar x))
        (t (cons var x))))

;;; Multiply two polynomials x and y
;;;   ptimes is immediately strict with respect to x, but
;;;   not strict with respect to y.  It is not quick.
(defun ptimes (x y)
  (cond ((or (pzerop x) (pzerop y)) 0)
        ((atom x) (pctimes x y))
        ((atom y) (pctimes y x))
        ((eq (car x) (car y))
         (psimp (car x) (ptimes1 (cdr x) (cdr y))))
        ((> (get (car x) 'order) (get (car y) 'order))
         (psimp (car x) (pctimes1 y (cdr x))))
        (t (psimp (car y) (pctimes1 x (cdr y))))))

;;; ptimes1 does the real work of multiplying two polynomials together.
;;; The concept: multiplying a polynomial by a "monomial" (a*x^m) is easy.
;;; Multiply by each of the monomials, and add it all together.

;;; Multiply two coefficient lists
;;;   ptimes is immediately strict with respect to x,
;;;   but not strict with respect to y.  It is not quick.
;;;   put futures around cons's arguments, since it is not strict
(defun ptimes1 (x y)
  (cond ((null x) nil)
        ((null y) nil)
        (t (pplus1 (future (do ((y y (cdr y))
                                (result nil))
                               ((null y) (nreverse result))
                             (setq result
                                   (cons (cons (future (pplus (caar x)
                                                              (caar y)))
                                               (future (ptimes (cdar x)
                                                               (cdar y))))
                                         result))))
                   (future (ptimes1 (cdr x) y))))))

;;; Raise polynomial p to the nth power.
;;;   pexptsq is immediately strict with respect to n, but
;;;   not strict with respect to p.  It is not quick.
(defun pexptsq (p n)
  (if (oddp n)
      (if (= n 1)
          p
          (ptimes p (pexptsq (future (ptimes p p)) (floor n 2))))
        (if (zerop n)
          1
          (pexptsq (future (ptimes p p)) (floor n 2)))))

(eval-when (compile load eval)
  (setf (get 'x 'order) 1)
  (setf (get 'y 'order) 2)
  (setf (get 'z 'order) 3))
```

APPENDIX B

The Code

B.1. analyze.lisp

```
;;;; analyze.lisp
;;;;
;;;; This file contains the functions for the analysis phase.  It uses
;;;; functions in util.lisp

(in-package 'simpl)
(use-package 'flavors)
(export '(analyze))

(defvar *event-times*)      ;Association list of event numbers and the time
                   ; the event started
(defvar *ifd*)              ;Current input stream
(defvar *ofd*)              ;Current output stream
(defvar *sim*)              ;The main task
(defvar *seq*)              ;Sequence number, used in sorting
(defvar *infinity* 99)      ;The number of processors to allocate if an infinite
                   ; number are requested
(defvar *processors*)       ;An array of instances of the processor flavor
(defvar *call-stack*)       ;Stack of tasks called

(proclaim '(fixnum *seq* *infinity*))

;;; Add a new event-number, start-time pair to *event-times*
(defmacro add-new-event (number base-time)
  '(push (cons ,number ,base-time) *event-times*))

;;; Get the start time associated with an event number
(defmacro get-base-time (number)
  '(cdr (assoc ,number *event-times*)))

;;; Does an event number already have a start time stored?
(defmacro event-exists (number)
  '(assoc ,number *event-times*))

;;; Read something unimportant
(defmacro read-dummy ()
  '(read *ifd* nil nil))

;;; Read another event number
(defmacro read-number ()
  '(read *ifd* nil nil))

;;; Read another task name
(defmacro read-id ()
  '(read *ifd* nil nil))

;;; Read another event type
(defmacro read-type ()
  '(read *ifd* nil nil))

;;; Read another time
(defmacro read-time ()
  '(read *ifd* nil nil))

;;; Define a "processor" to be something which has a current task
;;; which it is "running", and a queue of tasks waiting to be run.
(defflavor processor
  ((current-task nil)
   (queue nil))
  ()
  :settable-instance-variables)

;;; Define an "event".
(defflavor event
  (id       ;The name of the task this event affects
   number   ;Events with the same number occur at the same time
   type             ;What kind of event this was
   time             ;When it happened, relative to the parent
   parent)  ;Mommy!
  ()
  :settable-instance-variables)

;;; Define a "task" object.
(defflavor task
  ((id nil)                 ;The task's "name"
   (event-number nil)             ;Its event number
   (parent-wait 0)          ;Amount of time its parent spent idle
                            ; before this task started
   (start nil)                    ;Time started
   (end nil)                ;Time finished
   (events nil)                   ;List of events which happened to this
                            ; task
   (idle-time 0)            ;Amount of time spent idle
   (last-time-blocked nil)        ;Time at which task was last stopped
   (parent nil)                   ;Task's parent
   (processor nil)          ;Processor we are running on
   (queue nil))                   ;Queue of other tasks waiting on our
                            ; result
  ()
  :settable-instance-variables)
```

```
;;; The "main start" time of a task refers to the actual time it
;;; starts in the multiprocessor environment, rather than the
;;; relative times which are stored.  This time is calculated by
;;; summing the "main start" time of the task's parent, the amount
;;; of time the parent spent idle before this task was started,
;;; and the time at which this task started relative to its parent's
;;; start.
(defmethod (task :main-start) ()
  (let ((parent (send self :parent)))
    (if parent
      (+ (the fixnum (send self :start))
         (the fixnum (send self :parent-wait))
         (the fixnum (send parent :main-start)))
      (send self :start))))

;;; The "main end" time of a task refers to the actual time it
;;; ends in the multiprocessor environment, rather than the
;;; relative times which are stored.  This time is calculated by
;;; summing the "main start" time of the task's parent, the amount
;;; of time the parent spent idle before this task was started,
;;; and the time at which this task ended relative to its parent's
;;; start.
(defmethod (task :main-end) ()
  (+ (the fixnum (send self :end))
     (the fixnum (send self :idle-time))
     (the fixnum (send self :main-start))))

;;; Set the time this task was last blocked.  This is calculated
;;; by summing the "main start" time, the idle time, and the
;;; current time.
(defmethod (task :set-last-time-blocked) (time)
  (if time
      (setq last-time-blocked
           (+ (the fixnum (send self :main-start))
              (the fixnum (send self :idle-time))
              (the fixnum time)))
      (setq last-time-blocked nil)))

;;; Add some more idle time to the total already accumulated.
;;; The 'task' argument is the task whose ending or blocking
;;; caused the 'self' task to become active again.
(defmethod (task :update-idle-time) (task)
  (let ((last-time-active (if (send task :end)
                              (send task :main-end)
                              (send task :last-time-blocked))))
    (declare (fixnum last-time-active))
    (send self :set-idle-time
         (+ (the fixnum (send self :idle-time))
            (the fixnum (- last-time-active
                           (the fixnum (send self :last-time-blocked))))))))
```

```
;;; Define a macro to call the analyzer so that quoting
;;; is not necessary.  The 'ifile' argument should be a
;;; file previously created by the 'sim' function.  If
;;; not specified, 'ofile' will default to the current
;;; standard output, and 'nprocs' will default to "infinity"
;;; (which is 99 -- this is not a very big universe)
;;; Example:  (analyze sim.results sim.plot 3)
(defmacro analyze (ifile &optional (ofile *standard-output*)
                                    (nprocs *infinity*))
  '(analyze-func ',ifile ',ofile ,nprocs))

;;; Invoked to analyze an entire simulation.  It initializes
;;; global variables, creates the processors, and calls the
;;; functions which do the analysis.  Not meant to be called
;;; directly, but it won't hurt anything if you do.
(defun analyze-func (ifile ofile nprocs)
  (declare (special nprocs))
  (declare (fixnum nprocs))
  (if (<= nprocs 0)
      (error "Must have more than 0 processors!"))
  (setq      *processors* (make-array nprocs)
        *event-times* nil
      *sim* nil
      *call-stack* nil
      *seq* 0)
  (dotimes (i nprocs) (setf (aref *processors* i) (make-instance 'processor)))
  (with-open-file (*ifd* ifile :direction :input)
    (analyze-buildtree nil 0))
  (send (aref *processors* 0) :set-current-task *sim*)
  (send *sim* :set-processor (aref *processors* 0))
  ; Get the next event from each processor, and choose the one which would
  ; occur first
  (let ((tfile (format nil "~A.~D.TMP" ifile nprocs)))
    (with-open-file (*ofd* tfile :direction :output)
      (log-event 0 (send *sim* :id) 'begin-sim 0)
      (do ((pair (choose-next-task) (choose-next-task)))
        ((null pair))
      (if (car pair) (apply #'do-event pair))))
    (sortfile tfile tfile)
    (with-open-file (*ifd* tfile :direction :input)
      (with-open-file (*ofd* ofile :direction :output)
      (analyze-output)))))

;;; Invoked when a new task is begun.  Returns the
;;; end time of the task.  It builds a tree with
;;; the main process as the root, and its subprocesses
;;; branches.
(defun analyze-buildtree (oid base-time)
  (declare (fixnum base-time))
  (do ((number (read-number) (read-number))
       (id (read-id) (read-id))
```

```
          (type (read-type) (read-type))
          (time (read-time) (read-time))
          (offset 0))
         ((eq oid id)
          (if id (create-event number id type (- time base-time offset)))
          time)
       (declare (fixnum number time offset))
       (case type
         ((begin-exec)
          (cond ((event-exists number)
                 (create-event number
                               id
                               type
                               (get-base-time number)))
;                              (- (the fixnum (get-base-time number))
;                                 base-time)))
                (t
                 (add-new-event number (- time base-time offset))
                 (create-event number id type (- time base-time offset))))
          (setq offset
                (+ offset
                 (the fixnum
                      (- (the fixnum (analyze-buildtree id time))
                       time)))))
         ((begin-sim value-needed end-sim)
          (if (eq type 'begin-sim)
            (setq base-time time))
          (create-event number id type (- time base-time offset)))
         (t nil)))) ; do nothing if not one of the above types

;;; This function takes the information produced by 'analyze-buildtree'
;;; and converts it to x-y pairs, with x being time and y being
;;; the number of processors in use.  It writes this information to
;;; the output file specified in the call to 'analyze'.
(defun analyze-output ()
  (declare (special nprocs))
  (do ((last-x -1 x)
       (max-y 0)
       (area 0)
       (number (read-number) (read-number))
       (id (read-id) (read-id))
       (type (read-type) (read-type))
       (x (read-time) (read-time))
       (dummy (read-dummy) (read-dummy))
       (y 0))
      ((null x)
       (format *ofd* "~D ~D~%" last-x 0)
       (format *ofd* "; Maximum of ~D processor~:P in use~%" max-y)
       (format *ofd* "; Processor time used:  ~D~%" area)
       (if (not (= nprocs *infinity*))
         (setq max-y nprocs))
```

```
        (format *ofd*
              "; Total processor time available:  ~D~%" (* last-x max-y))
        (format *ofd*
              "; Percentage used:  ~,2F~%" (/ (float (* 100 area))
                                              (float (* last-x max-y)))))
    (declare (fixnum x y last-x max-y area))
    (declare (ignore number dummy))
    (cond ((= x last-x)
           (case type
             ((begin-sim begin-exec restarted)
              (incf y))
             ((end-sim end-exec blocked)
              (decf y))))
          (t
           (case type
             ((begin-exec restarted)
              (format *ofd* "~D ~D~%" last-x y)
              (format *ofd* "~D ~D~%" x y)
              (incf area (the fixnum (* y (the fixnum (- x last-x)))))
              (setq max-y (max y max-y))
              (incf y))
             ((end-exec blocked)
              (format *ofd* "~D ~D~%" last-x y)
              (format *ofd* "~D ~D~%" x y)
              (incf area (the fixnum (* y (the fixnum (- x last-x)))))
              (setq max-y (max y max-y))
              (decf y))
             ((begin-sim)
              (incf y)
              (setq max-y y)
              (format *ofd* "~D ~D~%" x 0))
             ((end-sim)
              (format *ofd* "~D ~D~%" last-x y)
              (format *ofd* "~D ~D~%" x y)
              (incf area (the fixnum (* y (the fixnum (- x last-x)))))
              (setq max-y (max y max-y))
              (decf y))
             (t
              (setq x last-x))))))

;;; Create an event instance, and also a new task if warranted
;;; by the type of event.  Stuff the result in the process tree
;;; being built by analyze-buildtree.
(defun create-event (number id type time)
  (declare (fixnum number time))
  (case type
    ((begin-sim)
     (set id (make-instance 'task :id id :event-number number :start time))
     (setq *sim* (symbol-value id))
     (push '*sim* *call-stack*))
    ((begin-exec)
     (let* ((parent (symbol-value (car *call-stack*)))
            (events (send parent :events))
            (me (make-instance 'task
                               :id id
                               :event-number number
                               :start time
                               :parent parent))
            (newevent (make-instance 'event
                                     :parent parent
                                     :number number
                                     :id id
                                     :type type
                                     :time time)))
       (set id me)
       (send parent :set-events (nconc events (list newevent)))
       (push id *call-stack*)))
    ((end-exec end-sim value-needed)
     (let* ((parent (symbol-value (car *call-stack*)))
            (events (send parent :events))
            (newevent (make-instance 'event
                                     :parent parent
                                     :number number
                                     :id id
                                     :type type
                                     :time time)))
       (send parent :set-events (nconc events (list newevent)))
       (if (not (eq type 'value-needed))
         (pop *call-stack*))))
    (t (error "ERROR:  got ~D ~S ~S in create-event~%" number id type))))

;;; Choose one of the events returned by get-next-tasks as the
;;; next one to execute.  This is done by figuring out when each
;;; would occur in real time and choosing the one which would occur
;;; first.
(defun choose-next-task ()
  (car (sort (get-next-tasks)
             #'<
             :key #'(lambda (x)
                      (let ((event (car x))
                            (parent (send (car x) :parent)))
                       (+ (the fixnum (send event :time))
                          (the fixnum (send parent :main-start))
                          (the fixnum (send parent :idle-time))))))))

;;; Return a list of consisting of the next event
;;; which would occur in each active process.
(defun get-next-tasks ()
  (declare (special nprocs))
  (do ((i 0 (1+ i))
       (result nil))
      ((>= i nprocs) result)
    (declare (fixnum i))
    (let* ((this-processor (aref *processors* i))
           (current-task (send this-processor :current-task))
           (event (if current-task (first (send current-task :events)))))
      (if (and current-task event)
        (setq result (cons (list event this-processor) result))))))

;;; Take an event, plus the current processor, and make the
;;; changes which would result from the occurence of the
;;; event.
(defun do-event (next-event processor)
  (declare (special nprocs))
  (let* ((current-task (send processor :current-task))
         (events (send current-task :events))
         (number (send next-event :number))
         (id (send next-event :id))
         (type (send next-event :type))
         (time (send next-event :time)))
    (declare (fixnum number time))
    (case type
      ((begin-exec)
       (do ((i 0 (1+ i))
            (found nil))
           ((or found (>= i nprocs))
            (cond ((not found)
                   (send processor
                         :set-queue (cons current-task
                                          (send processor :queue)))
                   (send current-task :set-last-time-blocked time)
                   (send current-task :set-processor nil)
                   (send (symbol-value id)
                         :set-parent-wait (send current-task :idle-time))
                   (log-event '-
                              (send current-task :id)
                              'blocked
                              (send current-task :last-time-blocked))
                   (send processor :set-current-task (symbol-value id))
                   (send (symbol-value id) :set-processor processor))))
         (declare (fixnum i))
         (let ((proc (aref *processors* i)))
           (cond ((not (send proc :current-task))
                  (send proc :set-current-task (symbol-value id))
                  (send (symbol-value id) :set-processor proc)
                  (send (symbol-value id)
                        :set-parent-wait (send current-task :idle-time))
                  (setq found t)))))
       (log-event number id type (send (symbol-value id) :main-start))
       (send current-task :set-events (cdr events)))
      ((end-exec end-sim); We're finished
       ; Set the ending time of this task, adding in the wait time
       (send current-task :set-end time)
  ; This task is no longer running on any processor
  (send current-task :set-processor nil)
  ; Print self
  (log-event '- id type (send current-task :main-end))
  ; Pop the event queue
  (send current-task :set-events (cdr events))
  ; Empty the processor
  (send processor :set-current-task nil)
  ; Activate any tasks that might have been waiting for a value from
  ; this task
  (activate-future-queue current-task processor)
  ; If there weren't any, get one from one of the processor queues
  (if (null (send processor :current-task))
    (send processor
         :set-current-task (get-idle-task processor
                                          current-task)))
  ; If we found one, tell it where it's running
  (cond ((send processor :current-task)
        (send (send processor :current-task) :set-processor processor))))
 ((blocked-on-value)
  (let ((my-main-time (+ time
                   (the fixnum (send current-task :idle-time))
                   (the fixnum (send current-task :main-start))))
      (its-end-time (if (send (symbol-value id) :end)
                    (send (symbol-value id) :main-end))))
   (declare (fixnum my-main-time its-end-time))
   (cond (its-end-time
        (cond ((<= its-end-time my-main-time)
               (log-event '- id 'value-received my-main-time))
              (t
               (log-event
              '- id 'value-received
              (+ time
                 (the fixnum (send current-task :idle-time))
                 (the fixnum (send current-task :main-start))
                 (the fixnum (- its-end-time my-main-time))))
               (send current-task
                    :set-idle-time
                    (+ (the fixnum (- its-end-time my-main-time))
                     (the fixnum (send current-task
                                       :idle-time))))))
        (send current-task :set-events (cdr events)))
         (t
        (let ((proc (send current-task :processor)))
          (send (symbol-value id)
               :set-queue (cons current-task
                               (send (symbol-value id) :queue)))
          (send current-task :set-last-time-blocked time)
           (log-event '-
                    (send current-task :id)
                    'blocked
```

```
                     (send current-task :last-time-blocked))
              (send current-task :set-processor nil)
              (send proc :set-current-task
                   (get-any-idle-task current-task))
              (if (send proc :current-task)
                   (send (send proc :current-task)
                        :set-processor proc))))))
       ((value-needed)
        (let ((my-main-time (+ time
                             (the fixnum (send current-task :idle-time))
                             (the fixnum (send current-task :main-start))))
             (its-end-time (if (send (symbol-value id) :end)
                              (send (symbol-value id) :main-end))))
         (declare (fixnum my-main-time its-end-time))
         (log-event number id type my-main-time)
         (cond (its-end-time
                (cond ((<= its-end-time my-main-time)
                       (log-event '- id 'value-received my-main-time))
                      (t
                       (log-event
                      '- id 'value-received
                      (+ time
                          (the fixnum (send current-task :idle-time))
                          (the fixnum (send current-task :main-start))
                          (the fixnum (- its-end-time my-main-time))))
                       (send current-task
                            :set-idle-time
                            (+ (the fixnum (- its-end-time my-main-time))
                              (the fixnum (send current-task
                                                 :idle-time))))))
                (send current-task :set-events (cdr events)))
               (t
                (let ((proc (send current-task :processor)))
                  (log-event '- (send current-task :id) 'blocked my-main-time)
                  (send (symbol-value id)
                       :set-queue (cons current-task
                                        (send (symbol-value id) :queue)))
                  (send current-task :set-last-time-blocked time)
                  (send next-event :set-type 'blocked-on-value)
                  (send current-task :set-processor nil)
                  (send proc :set-current-task
                       (get-any-idle-task current-task))
                  (if (send proc :current-task)
                      (send (send proc :current-task)
                           :set-processor proc))))))))
       (t (error "ERROR:  got ~D ~S ~S in do-event~%" number id type)))))

;;; Activate as many tasks as possible from the queue waiting for
;;; a particular future (specified by the 'task' argument) to be
;;; determined.  Push the rest onto the queue of the argument processor.
(defun activate-future-queue (task processor)
  (declare (special nprocs))
  (do ((i 0 (1+ i))
       (done nil))
      ((or done (>= i nprocs)))
    (declare (fixnum i))
    (let* ((proc (aref *processors* i))
           (current-task (send proc :current-task)))
      (cond ((null current-task)
             (let* ((q (send task :queue))
                    (newtask (car q)))
               (cond (newtask
                      (send proc :set-current-task newtask)
                      (send newtask :set-processor proc)
                      (send newtask :update-idle-time task)
                      (send newtask :set-last-time-blocked nil)
                      (log-event '-
                                 (send newtask :id)
                                 'restarted
                                 (send task :main-end))
                      (send task :set-queue (cdr q)))
                     (t
                      (setq done t))))))))
  ;; If there are any tasks left in the queue, stick them on the
  ;; current processor's queue.
  (cond ((send task :queue)
         (send processor
               :set-queue (nconc (send task :queue)
                                 (send processor :queue)))
         (send task :set-queue nil))))

;;; Pop an idle task off a particular processor's queue.
;;; If the queue is empty, get a task from some other
;;; processor's queue.  The 'oldtask' argument is passed in
;;; order to access its end time, which is used as the new
;;; task's start time.
(defun get-idle-task (proc oldtask)
  (let* ((q (send proc :queue))
         (newtask (car q)))
    (cond (q
           (send proc :set-queue (cdr q))
           (send newtask :update-idle-time oldtask)
           (log-event '- (send newtask :id) 'restarted
                      (if (send oldtask :end)
                         (send oldtask :main-end)
                         (send oldtask :last-time-blocked)))
           (send newtask :set-last-time-blocked nil)
           newtask)
          (t
           (get-any-idle-task oldtask)))))

;;; Pop an idle task off some processor's queue.  Return nil if
;;; all queues are empty.  The 'oldtask' argument is passed in
;;; order to access its end time, which is used as the new
;;; task's start time.
(defun get-any-idle-task (oldtask)
  (declare (special nprocs))
  (do* ((i 0 (1+ i))
       (newtask nil))
      ((or newtask (>= i nprocs)) newtask)
    (declare (fixnum i))
    (let* ((proc (aref *processors* i))
           (q (send proc :queue)))
      (cond (q
             (setq newtask (car q))
             (send newtask :update-idle-time oldtask)
             (send newtask :set-last-time-blocked nil)
             (log-event '-
                        (send newtask :id)
                        'restarted
                        (if (send oldtask :end)
                            (send oldtask :main-end)
                            (send oldtask :last-time-blocked)))
             (send proc :set-queue (cdr q)))))))

;;; Log some info about an event:  the event number, its name,
;;; its type, when it occurred, and a sequence number to keep
;;; it in the proper order after sorting, times being equal.
(defun log-event (number id type time)
  (declare (fixnum number time))
  (format *ofd* "~D ~S ~S ~D ~D~%" number id type time (incf *seq*)))
```

B.2. ehook.lisp

```
;;;; ehook.lisp
;;;;
;;;; This file contains the functions which are required for
;;;; use of the *evalhook* mechanism.  It uses functions defined
;;;; in util.lisp

(in-package 'simpl)
(export '(future delay pcall touch))

(defvar *depth-stack*)          ;A stack whose first element is the
                          ; current depth inside an expression
(defvar *total-delays*)         ;The total number of delays created so far.
(defvar *uncomputed-delays*)    ;The number of delays whose values have not
                          ; yet been required.
(defvar *total-futures*) ;The total number of futures created so far.
(defvar *unneeded-futures*)     ;The number of futures whose values have not
                          ; yet been required.
```

```
(defvar *call-stack*)           ;A stack of the function names called
(defvar *event-level*)          ;A stack of the current event number
(defvar *future-levels*)  ;An association list of event numbers, and an
                          ; atom which shows whether the last expression
                          ; at that level was a future or not.

(proclaim '(fixnum *total-delays*
                 *uncomputed-delays*
                 *total-futures*
                 *unneeded-futures*))

;;; Check to see if a function is strict by looking for the
;;; atom 'strict' on its property list.  Strict functions
;;; are defined in sim.lisp
(defmacro strict-p (f)
  '(get ,f 'strict))

;;; Check to see if a object is a symbol representing
;;; a future by looking for the atom 'future' on its
;;; property list.
(defmacro future-p (x)
  '(and (symbolp ,x)
      (get ,x 'future)))

;;; Check to see if a object is a symbol representing
;;; a delay by looking for the atom 'delay' on its
;;; property list.
(defmacro delay-p (x)
  '(and (symbolp ,x)
      (get ,x 'delay)))

;;; Define a structure to contain the closure created for a
;;; delay.  The purpose of defining the structure is merely
;;; for ease of identification of a delay.
(defstruct (delay-struct (:conc-name delay-))
  closure)

;;; Expects a symbol whose value is a delay structure, or the
;;; value resulting from its previous computation.  Returns
;;; that value, or computes it if necessary.  A side effect is
;;; that the symbol's value will be set to the result.
(defun do-delay (expr)
  (cond ((delay-struct-p (symbol-value expr))
        (decf *uncomputed-delays*)
        (logmsg (next-event) expr 'begin-delay)
        (set expr (funcall (delay-closure (symbol-value expr))))
        (logmsg nil expr 'end-delay)
        (symbol-value expr))
       (t
        (logmsg (next-event) expr 'already-computed)
        (symbol-value expr))))
```

```
;;; Expects a symbol whose value is the result of a future.
;;; If this is the first time it has been requested, the
;;; 'determined' property is set to TRUE.
;;; Returns the value, or if the value turns out to be a future
;;; also, recurses.
(defun do-future (f)
  (cond ((not (get f 'determined))
         (setf (get f 'determined) t)
         (decf *unneeded-futures*)))
  (logmsg (next-event) f 'value-needod)
  ; Recursively check the value to see if it also is a future.
  (let ((f1 (symbol-value f)))
    (if (future-p f1)
      (do-future f1)
      f1)))

;;; This is the function which gets control of the evaluation process
;;; from eval, using the *evalhook* variable.  It takes a form to
;;; evaluate, plus an optional environment, which is used by Commcn
;;; LISP in determining bindings.  It returns the result of evaluating
;;; the form.
(defun ehook (form &optional env)
  (push (1+ (the fixnum (car *depth-stack*))) *depth-stack*)
  (let* ((depth (car *depth-stack*))
         (future-level (assoc depth *future-levels*))
         (values nil))
    ; Do the bookkeeping stuff to assure that we know what the last
    ; expression seen at this level was.
    (cond ((and (listp form) (eq 'future (car form)))
           (cond ((eq 'future (caddr future-level))
                  (setq *event-level*
                        (member (cadr future-level) *event-level*)))
                 (t
                  (push (next-event) *event-level*)
                  (push (list depth (car *event-level*) 'future)
                        *future-levels*))))
          (t
           (push (list depth (car *event-level*) 'non-future)
                 *future-levels*)))
    ; Check to see what type 'form' is.
    (cond ((listp form)
           ; It's a list -- must be a parallel construct or a
           ; function call
           (case (car form)
             (future
              (let ((sym (gentemp "future" *simpl-package*)))
               (setf (get sym 'future) t)
               (incf *total-futures*)
               (incf *unneeded-futures*)
               (logmsg (car *event-level*) sym 'begin-exec)
               (set sym (ehook (cadr form) env))
  (logmsg nil sym 'end-exec)
  ; If we know the outer function is strict, better
  ; return the real value instead of the future symbol.
  ; Hopefully no one would create a future in this
  ; position anyway.
  (if (strict-p (car *call-stack*))
      (setq values (do-future sym))
      (setq values sym))))
(delay
   (let ((sym (gentemp "delay" *simpl-package*)))
  (setf (get sym 'delay) t)
  (incf *uncomputed-delays*)
  (incf *total-delays*)
  ;; Create a list of the atoms in 'form', so that
  ;; a closure can be made of their bindings
  (set sym
       (make-delay-struct
        :closure (ehook '(function
                      (lambda ()
                            (let ((*evalhook* 'ehook))
                                 ,(cadr form))))
                     env)))
  (logmsg (next-event) sym 'created)
  ; If we know the outer function is strict, better
  ; return the real value instead of the delay symbol.
  (if (strict-p (car *call-stack*))
      (setq values (do-delay sym))
      (setq values sym))))
(pcall
 (do ((elements (cdr form) (cdr elements))
     (newfunc nil)
     (newargs nil))
    ((null elements) (setq values (apply newfunc newargs)))
  (let ((sym (gentemp "future" *simpl-package*)))
    (setf (get sym 'future) t)
    (logmsg (car *event-level*) sym 'begin-exec)
    (set sym (ehook (car elements) env))
    (logmsg nil sym 'end-exec)
    (if (eq (car elements) (cadr form))
        (setq newfunc (symbol-value sym))
        (setq newargs
             (append newargs (list (symbol-value sym))))))))
(touch
 (setq values (ehook (cadr form) env))
 (if (delay-p values)
    (setq values (do-delay values)))
 (if (future-p values)
    (setq values (do-future values))))
(t
 ; It must be a regular function call.
 (push (car form) *call-stack*)
```

```
            (setq values (evalhook form #'ehook nil env))
            (setq *call-stack*
                (cdr (member (car form) *call-stack* :test #'equal)))
            (if (and (delay-p values)
                     (strict-p (car *call-stack*)))
              (setq values (do-delay values)))
            (if (and (future-p values)
                     (strict-p (car *call-stack*)))
              (setq values (do-future values)))))
        ; If we're evaluating an argument to a strict function,
        ; and it's a delay or future symbol, better return the
        ; real value.
        ((and (delay-p form) (strict-p (car *call-stack*)))
         (setq values (do-delay form)))
        ((and (future-p form) (strict-p (car *call-stack*)))
         (setq values (do-future form)))
        ; Just an ordinary variable.
        (t
         (setq values (evalhook form #'ehook nil env))
         (if (and (delay-p values)
                  (strict-p (car *call-stack*)))
             (setq values (do-delay values)))
         (if (and (future-p values)
                  (strict-p (car *call-stack*)))
             (setq values (do-future values)))))
    (setq *future-levels*
        (member depth *future-levels*
              :test #'(lambda (x y) (equal x (car y)))))
    (pop *depth-stack*)
    values))
```

B.3. main.lisp

```
(in-package 'simpl)

(setq *simpl-package* *package*)

(cond ((member :excl *features*) (setq *lisp-type* :excl))
      ((member :lucid *features*) (setq *lisp-type* :lucid))
      ((member 'common *features*) (setq *lisp-type* :common-lisp))
      ((member :common *features*) (setq *lisp-type* :common-lisp))
      ((member :common-lisp *features*) (setq *lisp-type* :common-lisp))
      (t (error "What kind of lisp is this, anyway?")))

(case *lisp-type*
  ((:excl)
   (use-package 'excl))
  (t
   (use-package 'system)))
```

```
(load "util")
(load "ehook")
(load "sim")
(load "analyze")
```

B.4. sim.lisp

```
;;;; sim.lisp
;;;;
;;;; This file contains the functions for the simulation phase.
;;;; It uses functions defined in util.lisp.

(in-package 'simpl)
(export '(sim))

;;; Mark certain functions as 'strict'.  Only a useful subset of the
;;; strict functions in Common LISP are included here.
(eval-when (load eval)
  (mapc #'(lambda (x)
           (setf (get x 'strict) t))
       '(listp not null if cond get atom numberp zerop
             + - * = eq equal eql
             > < >= <= 1+ 1- and or car cdr caar cadr
             caadr caaar cadar caddr
             cdar cddr cdaar cdadr cddar cdddr pprint
             print princ prin1 write write-to-string
             prin1-to-string princ-to-string write-char
             write-string write-line write-byte format oddp evenp)))

;;; Define a macro to call the simulator so that quoting
;;; is not necessary.  The first argument must be the form
;;; to be evaluated.  The second argument is the name of the
;;; file to write to.  If not specified, it defaults to
;;; whatever is currently defined as standard output.
;;; Example:  (sim (quicksort '(2 1)) results.dat)
(defmacro sim (form &optional (ofile *standard-output*))
  '(with-open-file (*ofd* ',ofile :direction :output)
     (sim-func ',form)))

;;; This function is the real simulator.  It takes only the
;;; form to be evaluated.  It initializes global variables,
;;; turns off garbage collection, and sets up the 'evalhook'
;;; mechanism.  It then evaluates the form, turns on garbage
;;; collection, and logs pertinent information.  It returns
;;; the result stripped of all futures and delays.
(defun sim-func (form)
  (let ((val nil)
        (sym1 (gentemp "sim" *simpl-package*)))
    ;; Initialize important globals.
    (setq *total-delays* 0
        *uncomputed-delays* 0
        *total-futures* 0
        *unneeded-futures* 0
        *call-stack* nil
        *depth-stack* '(0)
        *event-level* '(0)
        *future-levels* nil
        *max-event* 0)
    ;; Do a garbage collection, then turn it off if we know how.
    ;; This allows consistency when the programs are run many
    ;; times.
    (gc)
    #+lucid (gc-off)
    ;; Write out the form we are evaluating, just for the record.
    (format *ofd* "; ~S~%" form)
    ;; Log the fact that the simulation is beginning.
    (logmsg *max-event* sym1 'begin-sim)
    ;; Turn on the evaluation hook which allows us to capture control
    ;; from eval, and evaluate the form.
    (let ((*evalhook* 'ehook))
      (setq val (eval form)))
    ;; Log the fact that the simulation has ended.
    (logmsg nil sym1 'end-sim)
    ;; Turn garbage collection on again.
    #+lucid (gc-on)
    ;; Build up a tree containing no futures or delays, so we can
    ;; print it out as the 'answer'.
    (setq val (build-it val))
    ;; Log a bunch of stuff.
    (format *ofd*
          "; ~S~%" val)
    (format *ofd*
          "; ~D total futures, ~D futures computed but not needed~%"
          *total-futures* *unneeded-futures*)
    (format *ofd*
          "; ~D total delays, ~D delays not computed~%"
          *total-delays* *uncomputed-delays*)
    ;; Return the value.
    val))

;;; Go through a tree which may contain futures or delays and
;;; substitute their values.
(defun build-it (x)
  (cond ((future-p x) (build-it (symbol-value x)))
      ((delay-p x) (build-it (symbol-value x)))
      ((consp x) (cons (build-it (car x))
                   (build-it (cdr x))))
      (t x)))
```

B.5. util.lisp

```
;;;; util.lisp
;;;;
;;;; This file contains some utility functions shared by other parts
;;;; of the system.

(in-package 'simpl)

(defvar *max-event*)       ; The highest event number used so far

(proclaim '(fixnum *max-event*))

;;; Choose the function to use for timing
(eval-when (compile eval)
  (case *lisp-type*
    ((:lucid :excl :common-lisp)
     (defmacro cputime () '(get-internal-run-time)))
    (t
     (defmacro cputime () '(error "Cannot get cpu time.")))))

;;; Generate the next higher event number
(defun next-event ()
  (incf *max-event*))

;;; Write out a message giving the event number, the name of
;;; the task, the type of event, and the time at which it
;;; occurred.  If the event number is nil (meaning it is related
;;; to some other event), write '- instead.
(defun logmsg (event-number id type)
  (if event-number
      (format *ofd* "~D ~S ~S ~S~%" event-number id type (cputime))
      (format *ofd* "- ~S ~S ~S~%" id type (cputime))))

;;; Flatten a tree into a list.
(defun flatten (l)
  (cond ((null l) l)
        ((atom (car l)) (cons (car l) (flatten (cdr l))))
        (t (append (flatten (car l)) (flatten (cdr l))))))

;;; Use the UNIX sort utility to sort a file by the 4th and 5th fields,
;;; which should be integers.
(defun sortfile (ifile ofile)
  (case *lisp-type*
    ((:excl)
     (run-shell-command
      (format nil "sort +3n +4n -o ~A ~A" ofile ifile)))
    ((:lucid)
     (run-unix-program
      "sort" :arguments (list "+3n" "+4n" "-o" ofile ifile)))
    (t
    (error "Cannot sort file -- no way to run a subshell"))))
```

References

[1] Hal Abelson, Norman Adams, David Bartley, Gary Brooks, William Clinger, Dan Friedman, Robert Halstead, Chris Hanson, Chris Haynes, Eugene Kohlbecker, Don Oxley, Kent Pitman, Jonathan Rees, Bill Rozas, Gerald Jay Sussman and Mitchell Wand.
William Clinger, editor.
The Revised Revised Report on Scheme or an UnCommon Lisp.
Artificial Intelligence Memo 848, MIT Artificial Intelligence Laboratory, Cambridge, MA, August 1985.

[2] David Billstrom, Joseph Brandenburg and John Teeter.
CCLISP on the iPSC Concurrent Computer.
Intel Scientific Computers, Beaverton, OR, 1987.

[3] Melvin E. Conway.
Design of a Simple Transition-Diagram Compiler.
Communications of the ACM, VI(7):396-408, July 1963.

[4] Richard P. Gabriel.
Performance and Evaluation of Lisp Systems.
MIT Press, Cambridge, MA, 1985.

[5] Richard P. Gabriel and John McCarthy.
Queue-based Multi-processing Lisp.
ACM Symposium on LISP and Functional Programming, pages 25-43, ACM, August 1984.

[6] Sharon L. Gray.
Using Futures to Exploit Parallelism in Lisp.
Master's Thesis, Department of Electrical Engineering and Computer Science, MIT, February 1986.

[7] Robert H. Halstead, Jr.
Parallel Computing Using Multilisp.
MIT Laboratory for Computer Science, Cambridge, MA, 16 December 1986.

[8] Robert H. Halstead, Jr.
Multilisp: A Language for Concurrent Symbolic Computation.
ACM Transactions on Programming Languages and Systems, VII(4):501-538, October 1985.

[9] C. A. R. Hoare.
Parallel Programming: An Axiomatic Approach.
pp. 11-42 in *Language Hierarchies and Interfaces*, F. L. Bauer and K. Samelson, editors.
Springer-Verlag, New York, NY, 1976.

[10] P. Knueven, P. G. Hibbard and B. W. Leverett.
A Language System for a Multiprocessor Environment.
Robert B. K. Dewar, editor. *International Conference on the Design and Implementation of Algorithmic Languages*, pages 262-274, Courant Institute of

Mathematical Sciences, June 1976.

[11] Jed Marti and John Fitch.
The Bath Concurrent LISP Machine.
J. A. van Hulzen, editor. *Proceedings of EUROCAL '83*, pages 78-90, Springer-Verlag, March 1983.

[12] Patrick F. McGehearty and Edward J. Krall.
Potentials for Parallel Execution of Common Lisp Programs.
Kai Hwang, Steven M. Jacobs and Earl E. Swartzlander, editors. *Proceedings of the 1986 International Conference on Parallel Processing*, pages 696-702, IEEE Computer Society Press, August 1986.

[13] Curtis Alan Scott, Don Allen, Laura Bagnall, Jim Miller and Seth Steinberg.
Butterfly LISP Reference Manual.
BBN Laboratories Inc., April 1986.

[14] Shigeo Sugimoto, Kiyoshi Agusa, Koichi Tabata and Yutaka Ohno.
A Multi-microprocessor System for Concurrent LISP.
Proceedings of the 1983 International Conference on Parallel Processing, pages 135-143, IEEE Computer Society Press, June 1983.

[15] Stephen Michael Watt.
Bounded Parallelism in Computer Algebra.
Research Report CS-86-12, University of Waterloo, Waterloo, Ontario, Canada, May 1986.

[16] Benjamin Zorn, Paul Hilfinger, Kinson Ho, James Larus and Luigi Semenzato.
Features for Multiprocessing in SPUR Lisp.
University of California at Berkeley, in preparation, 26 September 1986.